



RAID 2020

PROCEEDINGS

**23rd International Symposium on
Research in Attacks, Intrusions
and Defenses**

October 14–16, 2020

**Proceedings of the
23rd International Symposium on
Research in Attacks, Intrusions and Defenses
(RAID 2020)**

October 14–16, 2020



**Mondragon
Unibertsitatea**

Gold Sponsor



Bronze Sponsors



© 2020 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-18-2

Organizing Committee

General Chair

Urko Zurutuza, *Mondragon Unibertsitatea*

Vice General Chairs

Enaitz Ezpeleta, *Mondragon Unibertsitatea*

Iñaki Garitano, *Mondragon Unibertsitatea*

Mikel Iturbe, *Mondragon Unibertsitatea*

Program Committee Chair

Manuel Egele, *Boston University*

Program Committee Co-Chair

Leyla Bilge, *NortonLifeLock Research Group*

Publication Chair

Igor Santos, *Mondragon Unibertsitatea*

Publicity Chair

Guillermo Suárez-Tangil, *King's College London*

Travel Grant Chair

Magnus Almgren, *Chalmers University of Technology*

Program Committee

Aisha Ibrahim Ali-Gombe, *Towson University*

Tiffany Bao, *Arizona State University*

Kevin Borgolte, *Princeton University*

Lorenzo Cavallaro, *King's College London*

Kai Chen, *Institute of Information Engineering,
Chinese Academy of Sciences*

Yaohui Chen, *Facebook*

Adrian Dabrowski, *University of California, Irvine*

Nathan Dautenhahn, *Rice University*

Lorenzo De Carli, *Worcester Polytechnic Institute*

Tobias Fiebig, *TU Delft*

Yanick Fratantonio, *Eurecom*

Carrie Gates, *Bank of America*

Luca Invernizzi, *Google, Inc.*

Vasilis Kemerlis, *Brown University*

Amin Kharraz, *University of Illinois at Urbana–Champaign*

Johannes Kinder, *Bundeswehr University Munich*

Andrea Lanzi, *University of Milan*

Nick Nikiforakis, *Stony Brook University*

Shirin Nilizadeh, *The University of Texas at Arlington*

Kaan Onarlioglu, *Akamai*

Jason Polakis, *University of Illinois at Chicago*

Georgios Portokalidis, *Stevens Institute of Technology*

Aravind Prakash, *Binghamton University*

William Robertson, *Northeastern University*

Brendan Saltaformaggio, *Georgia Institute of Technology*

Yun Shen, *NortonLifeLock Research Group*

Dave Tian, *Purdue University*

Fish Wang, *Arizona State University*

Zach Weinberg, *University of Massachusetts, Amherst*

Michael Weissbacher, *Square*

Chao Zhang, *Tsinghua University*

Yang Zhang, *CISPA*

Steering Committee

Johanna Amann, *International Computer Science Institute*

Davide Balzarotti, *Eurecom Graduate School and Research Center*

Michael Bailey, *University of Illinois at Urbana–Champaign*

Marc Dacier, *Eurecom Graduate School and Research Center*

Thorsten Holz, *Ruhr-Universität Bochum*

Zhiqiang Lin, *The Ohio State University*

Mathias Payer, *EPFL*

Michalis Polychronakis, *Stony Brook University*

Angelos Stavrou, *George Mason University*

External Reviewers

Babak Amin Azad

Timothy Barron

Min Chen

Matthew Cole

David Demicco

Rukayat Erinfolami

Panagiotis Ilia

Brian Kondracki

Christopher Lenk

Zheng Li

Yugeng Liu

Blake Loring

Fukutomo Nakanishi

Feergus Pendlebury

Hernán Ponce de León

Kostas Solomos

Flavio Toffalini

Shengdun Wang

Rui Wen

Yang Zou

Message from the RAID 2020 Program Co-Chairs

Welcome to the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)!

We are proud to present the proceedings of RAID 2020—of course, the ongoing pandemic has left its mark also on the RAID symposium and forced us to run the symposium in a virtual manner. Despite the additional effort this entails, especially on the general chair and his team, we are excited about the program that the program committee put together and we are happy to have received many excellent submissions.

This year, the symposium received 121 submissions, of which 31 were accepted (a 26% acceptance rate). As in previous years, the symposium adopted a double-blind reviewing process to ensure that the reviewers remained unaware of the authors' names and affiliations during review and subsequent discussions. Each paper received at least three reviews and the final decision for each paper was made during an 8-hour long online PC meeting in May 2020.

The quality and commitment of the Program Committee is paramount to the success of any conference. This year, we were fortunate to feature an international and diverse PC consisting of 36 members in total—33% of the PC members hailed from outside the US and, as has gotten customary for RAID, approximately 20% were from government, industry, or a mix. The PC included veteran PC members who have already served several times on the RAID PC, as well as many new members who served for the first time. We are grateful to the PC members for the effort they invested in selecting the best possible program from the pool of submissions.

In 2018, RAID introduced an annual Best Paper award. A subset of PC members were selected by the chairs and served as the award committee. Selected papers were discussed amongst the awards committee and then a vote amongst the committee decided the award winner. The winner will be announced during the opening session, stay tuned!

Starting last year, RAID switched to an open access license to publish the proceedings of the conference. Now for the second time, the proceedings are published by USENIX, and all papers are available online on the opening day of RAID 2020.

As a significant and positive aspect, USENIX allows authors to retain ownership of the copyright in their works, requesting only that USENIX be granted the right to be the first publisher of that work. We hope that this change will have a positive impact on the conference and the scientific community at large.

RAID only exists because of the community that supports it. Indeed, RAID is completely self-funded. Every organizer independently shoulders the financial risks associated with the symposium's organization. The sponsors, therefore, play an essential role and ensure that the registration fees remain very reasonable. To this end, we want to take this opportunity to thank our Gold Sponsor, the Basque Cybersecurity Centre (BCSC), and our Bronze Sponsors Aruba and ZIUR Foundation (Gipuzkoa's Industrial Cyber Security Agency) for their generous sponsorships to RAID 2020!

We would like to further express our gratitude to the general chair Urko Zurutuza from Mondragon Unibertsitatea and his vice general chairs Enaitz Ezpeleta, Iñaki Garitano, and Mikel Iturbe all from Mondragon Unibertsitatea. Our thanks also go to their assembled team of volunteers who will ensure the smooth operation of this year's virtual conference. Furthermore, we would like to thank the publication chair, Igor Santos from Mondragon Unibertsitatea, and the publicity chair Guillermo Suárez-Tangil from King's College London. Without their help, this conference could not have taken place, and their commitment was essential to host the symposium in the current virtual format.

Last, but not least, we want to thank all participants, authors, and attendees, who are of course the real heart and soul of the conference—thank you for making RAID such a wonderful conference!

We hope you enjoy the conference, please talk to us to provide feedback!

Manuel Egele and Leyla Bilge
RAID 2020 Program Committee Co-Chairs

23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)

October 14–16, 2020

Wednesday, October 14

Attacks

SpecROP: Speculative Exploitation of ROP Chains 1
Atri Bhattacharyya and Andrés Sánchez, *EPFL*; Esmail M. Koruyeh, Nael Abu-Ghazaleh, and Chengyu Song, *UC Riverside*; Mathias Payer, *EPFL*

Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners 17
Andrea Valenza, *University of Genova*; Gabriele Costa, *IMT School for Advanced Studies Lucca*; Alessandro Armando, *University of Genova*

Camera Fingerprinting Authentication Revisited 31
Dominik Maier, *Technische Universität Berlin*; Henrik Erb, Patrick Mullan, and Vincent Hupert, *Friedrich-Alexander-Universität Erlangen-Nürnberg*

Dynamic Program Analysis

Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities 47
Manh-Dung Nguyen and Sébastien Bardin, *Univ. Paris-Saclay, CEA LIST, France*; Richard Bonichon, *Tweag I/O, France*; Roland Groz, *Univ. Grenoble Alpes, France*; Matthieu Lemerre, *Univ. Paris-Saclay, CEA LIST, France*

WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS 63
Marcos Tileria and Jorge Blasco, *Royal Holloway, University of London*; Guillermo Suarez-Tangil, *King's College London, IMDEA Networks*

MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing 77
Yaohui Chen, Mansour Ahmadi, and Reza Mirzazade farkhani, *Northeastern University*; Boyu Wang, *Stony Brook University*; Long Lu, *Northeastern University*

Web Security

Tracing and Analyzing Web Access Paths Based on User-Side Data Collection: How Do Users Reach Malicious URLs? 93
Takeshi Takahashi, *National Institute of Information and Communications Technology*; Christopher Kruegel and Giovanni Vigna, *University of California, Santa Barbara*; Katsunari Yoshioka, *Yokohama National University*; Daisuke Inoue, *National Institute of Information and Communications Technology*

What's in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques 107
Ahmet Salih Buyukkayhan, *Microsoft*; Can Gemicioğlu, *Northeastern University*; Tobias Lauinger, *New York University*; Alina Oprea, William Robertson, and Engin Kirda, *Northeastern University*

Mininode: Reducing the Attack Surface of Node.js Applications 121
Igibek Koishybayev and Alexandros Kapravelos, *North Carolina State University*

Evaluating Changes to Fake Account Verification Systems 135
Fedor Kozlov, Isabella Yuen, Jakub Kowalczyk, Daniel Bernhardt, and David Freeman, *Facebook, Inc*; Paul Pearce, *Facebook, Inc and Georgia Institute of Technology*; Ivan Ivanov, *Facebook, Inc*

Thursday, October 15

Malware

SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub149
Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos, *UC Riverside*

HyperLeech: Stealthy System Virtualization with Minimal Target Impact through DMA-Based Hypervisor Injection 165
Ralph Palutke, Simon Ruderich, Matthias Wild, and Felix Freiling, *Friedrich-Alexander-Universität Erlangen*

Effective Detection of Credential Thefts from Windows Memory: Learning Access Behaviours to Local Security Authority Subsystem Service 181
Patrick Ah-Fat and Michael Huth, *Imperial College London*; Rob Mead, Tim Burrell, and Joshua Neil, *Microsoft*

Network & Cloud Security

EnclavePDP: A General Framework to Verify Data Integrity in Cloud Using Intel SGX 195
Yun He, *Institute of Information Engineering, Chinese Academy of Sciences, and School of Cyber Security, University of Chinese Academy of Sciences*; Yihua Xu, *Metropolitan College, Boston University*; Xiaoqi Jia, *Institute of Information Engineering, Chinese Academy of Sciences, and School of Cyber Security, University of Chinese Academy of Sciences*; Shengzhi Zhang, *Metropolitan College, Boston University*; Peng Liu, *Pennsylvania State University*; Shuai Chang, *Institute of Information Engineering, Chinese Academy of Sciences, and School of Cyber Security, University of Chinese Academy of Sciences*

Robust P2P Primitives Using SGX Enclaves 209
Yaoqi Jia, *ACM Member*; Shruti Tople, *Microsoft Research*; Tarik Moataz, *Aroki Systems*; Deli Gong, *ACM Member*; Prateek Saxena and Zhenkai Liang, *National University of Singapore*

aBBRate: Automating BBR Attack Exploration Using a Model-Based Approach 225
Anthony Peterson, *Northeastern University*; Samuel Jero, *Purdue University*; Endadul Hoque, *Syracuse University*; David Choffnes and Cristina Nita-Rotaru, *Northeastern University*

ML-Based Security

Cyber Threat Intelligence Modeling Based on Heterogeneous Graph Convolutional Network 241
Jun Zhao, *Beihang University*; Qiben Yan, *Michigan State University*; Xudong Liu, Bo Li, and Guangsheng Zuo, *Beihang University*

Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI 257
Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang, *Graph Computing Lab, George Washington University*

An Object Detection based Solver for Google's Image reCAPTCHA v2 269
Md Imran Hossen, Yazhou Tu, Md Fazle Rabby, and Md Nazmul Islam, *University of Louisiana at Lafayette*; Hui Cao, *Xi'an Jiaotong University*; Xiali Hei, *University of Louisiana at Lafayette*

Breaking ML

Evasion Attacks against Banking Fraud Detection Systems 285
Michele Carminati, Luca Santini, Mario Polino, and Stefano Zanero, *Politecnico di Milano*

The Limitations of Federated Learning in Sybil Settings 301
Clement Fung, *Carnegie Mellon University*; Chris J. M. Yoon and Ivan Beschastnikh, *University of British Columbia*

GhostImage: Remote Perception Attacks against Camera-based Image Classification Systems317
Yanmao Man and Ming Li, *University of Arizona*; Ryan Gerdes, *Virginia Tech*

Friday, October 16

CPS Security

PLC-Sleuth: Detecting and Localizing PLC Intrusions Using Control Invariants 333
Zeyu Yang, *Zhejiang University*; Liang He, *University of Colorado Denver*; Peng Cheng and Jiming Chen, *Zhejiang University*; David K.Y. Yau, *Singapore University of Technology and Design*; Linkang Du, *Zhejiang University*

Software-based Realtime Recovery from Sensor Attacks on Robotic Vehicles 349
Hongjun Choi and Sayali Kate, *Purdue University*; Yousra Aafer, *University of Waterloo*; Xiangyu Zhang and Dongyan Xu, *Purdue University*

SIEVE: Secure In-Vehicle Automatic Speech Recognition Systems 365
Shu Wang, *George Mason University*; Jiahao Cao, *George Mason University and Tsinghua University*; Kun Sun, *George Mason University*; Qi Li, *Tsinghua University and Beijing National Research Center for Information Science and Technology*

Firmware and Low Level Security

μ SBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability 381
Majid Salehi and Danny Hughes, *imec-Distrinet, KU Leuven*; Bruno Crispo, *imec-Distrinet, KU Leuven, and Trento University, Italy*

BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy Networks 397
Jianliang Wu, Yuhong Nan, and Vireshwar Kumar, *Purdue University*; Mathias Payer, *EPFL*; Dongyan Xu, *Purdue University*

Dark Firmware: A Systematic Approach to Exploring Application Security Risks in the Presence of Untrusted Firmware 413
Duha Ibdah, Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, and Hafiz Malik, *University of Michigan, Dearborn*

Systems Security

A Framework for Software Diversification with ISA Heterogeneity 427
Xiaoguang Wang, SengMing Yeoh, and Robert Lyerly, *Virginia Tech*; Pierre Olivier, *The University of Manchester*; Sang-Hoon Kim, *Ajou University*; Binoy Ravindran, *Virginia Tech*

Confine: Automated System Call Policy Generation for Container Attack Surface Reduction 443
Seyedhamed Ghavamnia and Tapti Palit, *Stony Brook University*; Azzedine Benameur, *Cloudhawk.io*; Michalis Polychronakis, *Stony Brook University*

sysfilter: Automated System Call Filtering for Commodity Software 459
Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis, *Brown University*

SpecROP: Speculative Exploitation of ROP Chains

Atri Bhattacharyya
EPFL

Andrés Sánchez
EPFL

Esmail M. Koruyeh
UC Riverside

Nael Abu-Ghazaleh
UC Riverside

Chengyu Song
UC Riverside

Mathias Payer
EPFL

Abstract

Speculative execution attacks, such as Spectre, reuse code from the victim’s binary to access and leak secret information during speculative execution. Every variant of the attack requires very particular code sequences, necessitating elaborate gadget-search campaigns. Often, victim programs contain few, or even zero, usable gadgets. Consequently, speculative attacks are sometimes demonstrated by injecting usable code sequences into the victim. So far, attacks search for monolithic gadgets, a single sequence of code which performs all the attack steps.

We introduce SpecROP, a novel speculative execution attack technique, inspired by classic code reuse attacks like Return-Oriented Programming to tackle the rarity of code gadgets. The SpecROP attacker uses multiple, small gadgets chained by poisoning multiple control-flow instructions to perform the same computation as a monolithic gadget. A key difference to classic code reuse attacks is that control-flow transfers between gadgets use speculative targets compared to targets in memory or registers.

We categorize SpecROP gadgets into generic classes and demonstrate the abundance of such gadgets in victim libraries. Further, we explore the practicality of influencing multiple control-flow instructions on modern processors, and demonstrate an attack which uses gadget chaining to increase the leakage potential of a Spectre variant, SMoTherSpectre.

1 Introduction

Spectre [1] demonstrated the power of speculative execution attacks by leaking information across various protection boundaries: sandboxes, processes, userspace/kernel, and virtual machines. These attacks reuse code gadgets (short instruction sequences with useful functionality) already present in the victim’s code base to access and leak secrets such as cryptographic keys or arbitrary memory. As a form of code-reuse attacks, they require gadgets composed of specific instruction sequences to exist in victim binaries. Often, the length of these

sequences, their complexity, or the rarity of their constituent instructions implies that the occurrence of usable gadgets is sparse. A case in point is the gadget required by Spectre. The attack requires a gadget where the attacker controls two registers, and contains two loads, of which the second load must access an address which depends on the value loaded by the first. In fact, the attack on the Linux kernel, which is a massive and diverse code base, relied on its eBPF (extended Berkeley Packet Filter) subsystem to essentially inject the gadget into the kernel.

To address the lack of powerful, monolithic gadgets, we propose the use of speculative gadget sequences. We present SpecROP, an attack based around the idea of using the effects of multiple, small gadgets to effectively perform computation equivalent to much larger, monolithic gadgets. The attack methodology leverages the relative abundance of the smaller gadgets as compared to larger gadgets to provide the required leakage gadgets. SpecROP is similar to existing code-reuse attacks (such as Return-Oriented Programming [2, 3] and Jump-Oriented Programming [4]). In comparison to these code reuse techniques, SpecROP leverages branch poisoning, a common starting step in speculative execution attacks, to effectively stitch the execution of these smaller gadgets. We search for small code sequences which perform common modifications on state (e.g., add, shift registers) and end in a return or indirect jump. During speculation, the CPU consults the branch predictor to decide the jump target, allowing us to redirect execution to the next gadget. We use automated binary analysis to discover the constituent gadgets of a SpecROP chain, using our tool, SpecFication.

This paper makes the following contributions:

- A study of the contexts in which branch targets may be maliciously influenced on modern processors (with hardware and microcode updates against Spectre-like attacks);
- Analysis of gadget chaining practicality, using indirect jump instructions as well as returns;
- A proof-of-concept attack, where we extend the capabilities of an existing speculative execution attack;

- A practical attack on a real target, `libcrypto` from OpenSSL, leaking multiple bits of the plaintext during encryption;
- A binary analysis tool, `SpecFication`, for discovering gadgets in real-world libraries, and a characterization of the existence of some classes of generic gadgets in commonly-used libraries.

2 Background

SpecROP extends the power and impact of speculative execution attacks [1, 5, 6] by enabling the combined use of multiple gadgets, similar in spirit as Return-Oriented Programming which enabled complex code-reuse attacks. Here, we provide the necessary background for SpecROP.

2.1 Speculative Execution Attacks

Modern processor design has led to a class of attacks known as Speculative Execution Attacks (SEA). These attacks target mechanisms designed to allow a processor to ameliorate the impact of long latency instructions on performance. Specifically, processors fetch and execute instructions out-of-order and speculate when lacking all the information needed to make decisions. Instructions executed by the processor following a misprediction are incorrect. While processors revert the architecturally visible effects of incorrect actions, microarchitectural side-effects remain. This allows SEA attacks to leak information encoded into cache residency [1, 5], or port utilization [6] during the period of incorrect execution. The period starting from the point the processor mispredicts until it realizes its mistake is the *speculation window*.

The microarchitectural structure, and its behavior which encodes information defines a side-channel. Variants of SEA differ in their choice of side-channel and the reason for misspeculation. Spectre-v1 [1] exploits misspeculation following a bounds-check prior to an array access. Spectre-v2 exploits misprediction of the target of an indirect call or jump. Both of these variant use a Flush+Reload channel [7]. SMOtherSpectre [6] and NetSpectre [8] use alternate side-channels based in port contention (ports are microarchitectural structures used for scheduling instructions within the processor pipeline) and the power-up status of AVX units instead.

2.2 Microarchitectural Side-channels

Microarchitectural side-channels are data channels on a processor which leverage state stored in microarchitectural structures, such as the cache or branch predictors, to transfer information. As opposed to a covert-channel, the transmitter in a side-channel is not privy to the communication: the transmitter is inadvertently leaking information and is referred to as the victim. The receiver reads the information from the channel, and is referred to as the attacker.

SEA depend on side-channels to leak any secrets accessed during speculative execution, since any architectural channels are erased when the processor detects misspeculation, and rolls back architecturally visible state. Here, we focus on two side-channels: a cache residency based channel (Flush+Reload), and a port contention based channel (SMoTher).

Flush+Reload channel A Flush+Reload channel [7] encodes information in the cache residency of a cache block at a specific address (A). This channel requires the attacker and victim to temporally share a core and its cache. Initially, the attacker primes the channel by flushing the cache block, evicting it from all layers of caches: the block is now uncached. In the second step, the victim executes. During its execution, the victim encodes a secret bit into the channel by conditionally loading from the address A . If the secret is 0, it loads A (thereby caching the block); if the secret is 1, it does not. Finally, the attacker reads the channel by reloading the address A , and timing how long it takes. If the secret was 0, and the victim had already cached the block, the attacker’s load is fast, else it is slow. This channel encodes a single bit.

A variant of this channel uses 256 unique address (A_0 through A_{255}) which map to different cache blocks. The victim encodes a secret byte (b) by only loading A_b . In the reload phase, the attacker reloads all addresses and times each load. The load A_b completes faster than the others, thereby leaking a byte.

SMoTher channel A SMOther channel [6] encodes information in the port utilization of the victim’s instructions. This side-channel requires the attacker and victim to share a Simultaneously Multi-Threaded (SMT) core, thereby sharing the ports on the core.

The victim encodes a secret bit by executing a SMOther gadget, a secret-dependent conditional branch leading to (target and fallthrough) code sequences which utilize different ports. Concurrently, the attacker executes a specific sequence of instructions designed to cause port contention with the target sequence, and times its execution. In the case that the victim is executing the fallthrough sequence, there is no port contention and the attacker completes its own sequence faster. In the other case, when the victim executes the target sequence, port contention causes the attacker’s execution to be slower. The attacker, therefore, uses its timing to infer which sequence the victim was executing. Since the victim’s secret bit determines which way the conditional branch goes, the attacker is able to leak the secret bit. This channel encodes a single bit.

2.3 Return/Jump Oriented Programming

Return-Oriented Programming (ROP) is a code-reuse technique based on chaining multiple instruction sequences al-

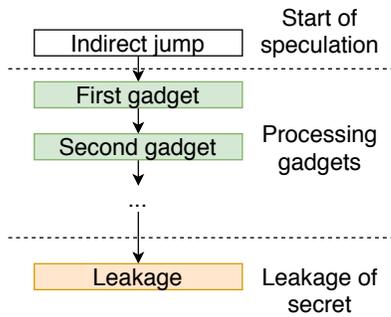


Figure 1: Phases of a Speculative ROP attack.

ready present in victim code into a gadget capable of performing complex computations. ROP gadgets end in a return instruction, chained through the attacker-compromised return addresses on the stack. This mechanism is used to chain the smaller sequences into complex exploits. In 2007, Shacham [2] demonstrated how gadgets from instruction sequences in `libc` could be used to achieve Turing-complete computation. Jump-Oriented Programming (JOP) [4] is a related technique where indirect jumps are hijacked through memory corruption to similarly compose smaller gadgets into complete attacks.

Similar to previously known buffer-overflow attacks, ROP requires the ability to corrupt the stack (or to pivot the stack to an alternate location). While buffer-overflow attacks were commonly used to inject and execute shell code onto the stack, they were effectively eliminated by one mitigation technique: hardware exclusion of writable and executable permissions on pages. ROP attacks bypass this mitigation by design, reusing instructions already existing within the victim’s binary and which must necessarily have executable permissions.

Code-reuse attacks depend on the existence of gadgets which perform computation useful to the attacker. The statistical probability of a gadget existing in a binary is dictated by two parameters: the length of the sequence and the probability of occurrence of each instruction in the gadget within the binary. On an architecture with variable-length instructions, such as `x86_64`, short instructions such as `ret` (encoded as one byte) may be even found inside the machine code for longer instructions (such as `mov rax, rbx`). In general, commonly existing ROP gadgets are small sequences of common, sometimes unintended, instructions.

3 Speculative ROP

Speculative Return-Oriented Programming (SpecROP), is an exploit technique that leverages gadget chaining to enhance the capabilities of an SEA attacker. Figure 1 shows the steps of a SpecROP attack. The attack starts at a mispredicted control-flow instruction, an indirect jump/call or return. The attacker poisons the branch predictor on the processor to control the

```

1 // C: array2[array1[x] * 4096]
2 mov (rax, rdi, 8), rax
3 shl 0xc, rax
4 mov (rdx, rax, 8), rax

```

(a) Spectre gadget where `rax` points to `array1`, `rdx` points to `array2`, and `rdi` is `x`

```

1 // Gadget 1: Load secret
2 mov (rdx, rax, 1), edx
3 call *(rbx + 0x40)
4 // Gadget 2: Shift secret
5 shl 0x20, rdx
6 mov eax, eax // Extraneous
7 or rdx, rax // Extraneous
8 ret
9 // Gadget 3: Leak secret
10 mov (rbx, rdx, 8), rsi

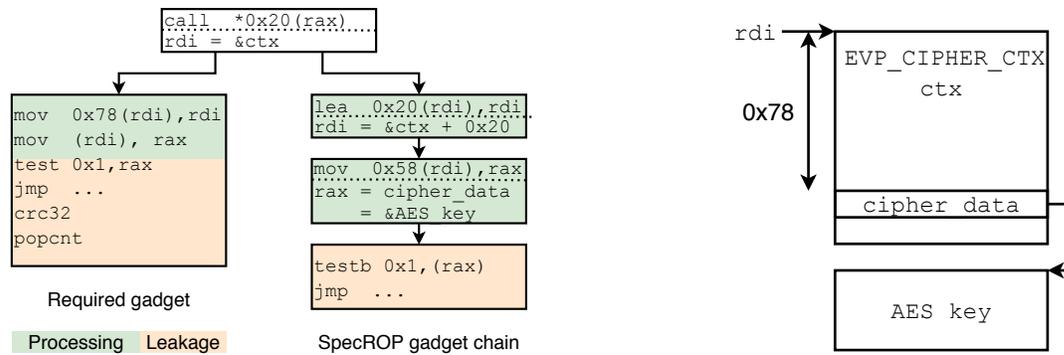
```

(b) Equivalent SpecROP chain where `rdx` points to `array1`, `rbx` points to `array2`, and `rax` is `x`. Lines 6 and 7 contain code irrelevant to the gadget chain.

Listing 1: A SpecROP chain from `libc` which can be used in place of the Spectre gadget

predicted target of the jump. Thereafter, the processor executes instructions along the first gadget in the chain: a *processing gadget*. Subsequently, the attacker manipulates a control-flow instruction at the end of the first gadget to direct execution to the second gadget, and so on through a gadget chain, similar to a ROP attack. The chain of processing gadgets is responsible for performing attacker-controlled computation, and is key to the increased capabilities of a SpecROP attacker (which we discuss later). The final gadget(s) in the chain, a *leakage gadget*, is used to leak secrets. *The key difference to ROP attacks is that SpecROP gadgets are executed and chained speculatively, i.e., the target of the indirect control flow transfer is not read from a memory location but indirectly influenced and “primed” by the attacker.*

SpecROP bypasses the reliance of existing SEA on two conditions which are hard to satisfy, by chaining and leveraging the execution of multiple gadgets. First, in classic SEA the victim’s secret needs to be directly accessible, either in a register or in memory referenced to by a register. Using a gadget chain overcomes this requirement, enabling attacks which require multiple operations to access the secret. For example, functions in the generic interface for the OpenSSL library (called EVP) have a pointer to a context structure as the first argument. The context includes a pointer to cipher-specific data (Figure 2b). For AES ciphers, this data is the encryption key. Accessing the key from the context pointer requires pointer arithmetic and two dereferences (see the “re-



(a) Control flow during an SEA attack, and for a SpecROP attack. (b) OpenSSL's Memory layout: The `EVP_CIPHER_CTX` structure contains a pointer to the AES key at an offset of `0x78`. Below the dotted lines, we show the relevant register state.

Figure 2: A SpecROP chain starting with a pointer to the OpenSSL cipher context structure, and capable of leaking the AES key.

quired gadget” in Figure 2a). Such a monolithic gadget which starts with a pointer to the context, and accesses and leaks the key requires a long sequence of instructions. We were unable to find such a gadget within multiple libraries. Existing SEA, therefore, are incapable of exploiting calls to the EVP functions to leak the key. In contrast, we found a chain using three gadgets from `libcrypto` (“SpecROP gadget chain” in Figure 2a) which allows attackers to access and leak the AES key. The full gadgets are listed in Appendix B. This chain requires the attacker to poison an extra indirect jump and an extra return instruction.

Second, classic SEA require a single gadget to both access the secret, and leak it into the microarchitectural channel. The gadget used to leak information in Spectre attacks requires two dependent memory loads with a left-shift of at-least 6 bits (to encode each value in a different, 64-byte cache line) in between (Listing 1a). So far, no natural gadget of this kind has been disclosed publicly, even for large, real-world binaries such as the Linux kernel. In fact, we applied our tool, SpecFication, to search for monolithic, Spectre-like gadgets in commonly-used libraries (listed in Section 5.2), the Linux kernel and its modules, and in QEMU without any results. In contrast, we found an equivalent SpecROP chain in `libc` (Listing 1b). The chain has three gadgets, two of which are dependent loads, with one shift gadget in between. Generally, each gadget in a SpecROP chain is shorter than a monolithic gadget, and there is a higher probability of finding them in the victim’s code.

SpecROP offers several benefits compared to traditional ROP and JOP attacks. Unlike ROP attacks, SpecROP does not require any corruption of the victim’s memory to chain gadgets. While there is an abundance of memory corruption attacks, they require the attacker to interact *directly* with the victim. In contrast, SpecROP attacks can be performed with only microarchitectural interactions between the attacker and their victim. Further, mitigations for memory safety which protect the stack state do not affect the SpecROP attacker. Un-

like JOP attacks, SpecROP does not require a dispatcher gadget whose repeated indirect calls enable chaining of gadgets. Finally, ROP/JOP attacks require that none of the chained gadgets have unwanted side-effects such as exceptions. By executing speculatively, SpecROP bypasses these restrictions. For example, it allows the transient execution of code that dereferences a null pointer as long as the loading of the secret or the leakage gadget do not depend on it.

Limitations SpecROP chains are limited in their length, in terms of the number of instructions and the number of cycles they take to execute. Processors have microarchitectural limits on the number of instructions they can fetch and execute speculatively: the re-order buffer can hold around 200 instructions on modern processors. The entire gadget chain must also complete its execution before the mispredicted branch is resolved. Following a last-level cache miss, the speculation window is up to a few hundred cycles [1].

Unlike JOP, SpecROP gadgets using indirect jumps cannot be repeated. The technique used to poison jumps (Branch Target Injection) implies a unique predicted target for each address, precluding the reuse of these gadgets in more than one place in the same chain.

3.1 Gadgets

Gadgets are the building blocks of a SpecROP attack, and an attacker will require the presence of a variety of gadgets to launch complicated attacks. We devise a system of categorization of gadgets in Section 3.1.1 and show metrics of the availability of each category in Section 4.

Gadgets in a typical SpecROP chain perform one of three important functions: (i) manipulate processor state to access secrets, and (ii) move said secrets into registers (iii) where a final gadget leaks them. At each step, one or more gadgets may be used. Figure 2 shows example gadgets from `libcrypto` which access and leak the key. The first gadget is an arithmetic

gadget, where the `lea` instruction effectively adds an offset to a pointer. The second gadget is a data movement gadget, and moves a pointer to the secret into register `rax`. Finally, the last gadget is a leakage gadget (using the SMOther side-channel) which dereferences a byte of the secret, and encodes the LSB into the channel.

3.1.1 Classification of gadgets

Gadgets in a SpecROP chain can be classified based on the functionality they provide. The main gadget categories are:

- *Arithmetic gadgets* perform simple arithmetic such as additions or subtractions. These might be useful for pointer manipulation, for example, allowing an attacker to craft pointers to secrets.
- *Shift gadgets* shift and rotate values in registers. Gadgets used in the SMOtherSpectre attack leak specific bits in registers (for example, the LSB). These gadgets allow an attacker to move secrets in other bits of the register into those bits which are leakable.
- *Data movement gadgets* move secrets or other data between registers, and between registers and memory. These can be used for moving secrets into a register targeted by the leakage gadget, for example.
- *Leakage gadgets* encode register/memory state into microarchitectural channels, enabling the attacker to later infer and leak information.
- *Multi-use gadgets* perform more than one of the above functions. An example is a `lea` gadget that performs an addition and multiplication.

This classification allows us to locate generic gadgets in existing code bases, rather than specific sequences for particular attack scenarios. In Section 5, we describe SpecFication, a tool for finding useful SpecROP gadgets, under constraints on their length, and starting and ending conditions.

4 Evaluation

We now evaluate the practical aspects of a SpecROP attack. First, we explore the contexts in which gadgets can be chained, and the limits on the number of control-flow instructions which can reliably be poisoned. Towards this goal, we explore both avenues of chaining SpecROP gadgets: indirect jumps and returns. Second, we create a prototype attack in a laboratory setting to explore how chained gadgets enhance a SMOtherSpectre attack, and whether there is any loss in accuracy of leaked information as a result of chaining gadgets. Finally, we describe a SpecROP attack on a real-world target, `libcrypto` from OpenSSL, demonstrating that such attacks are indeed feasible.

Context	6700K	8700	9700	10510U
Cross thread	Y	Y	N	N ¹
Cross process	N	N	N	N
Aliased	Y	Y	Y	Y

¹ even with factory microcode.

Table 1: Contexts in which branch poisoning is feasible

```

1 // Load unique address Ai
2 mov    (rdx), rcx
3 add    0x100, rdx
4 // Load address to next gadget
5 mov    (rdi), r8
6 add    0x8, rdi
7 // Jump to next gadget
8 jmpq   r8

```

Listing 2: Gadgets used to determine maximum chaining length using indirect jumps. The loads to A_i mark the execution of this gadget. The final jump chains to the next gadget.

4.1 Gadget chaining

The practicality of a SpecROP attack is strongly linked to the number of gadgets that can be reliably chained: the expressibility of the chain increases with the number of gadgets it contains. Most practical SEA attacks will require at least two (for the Spectre example) to three gadgets (for the OpenSSL example). We describe our experiments for chaining gadgets using indirect jumps and return instructions and evaluate the contexts under which we were able to influence the branch predictor.

Indirect jump poisoning in different contexts We evaluated the ability to poison the branch predictor

- across threads sharing an SMT physical core,
- across processes sharing an SMT physical core, and
- across instructions at different addresses, leveraging aliasing within the Branch Target Buffer (BTB).

We experimented with four generations of Intel’s processors with updated microcode: *i*) Skylake i7-6700K, *ii*) Coffee Lake i7-8700, *iii*) Coffee Lake Refresh i7-9700, and *iv*) Comet Lake i7-10510U. From Table 1, we can see that branch poisoning is only possible between an attacker and victim who share code execution within a process, for example, a browser sandbox running JavaScript from multiple websites.

4.1.1 Chaining gadgets through indirect jumps

Let us investigate the chaining of gadgets ending in indirect jumps and calls. Assuming that the targets for the jump are unavailable, the processor will use target predictions from

the BTB to speculatively fetch and execute instructions from multiple gadgets.

In our experiment, we execute two threads from the same process running on logical cores sharing a physical core. One of the threads takes the role of an attacker, using a sequence of indirect jumps through gadgets J_0 - J_{15} to train the branch predictor. Listing 2 shows the code for the gadgets. The other thread takes the role of the victim, speculatively following the same path through the gadgets. The goal of this experiment is to determine how many gadgets are actually executed by the victim.

The target for the terminal jump of each gadget is loaded from an array in memory (line 5), allowing us to “program” different paths for the attacker and victim. On the attacker, the gadgets are chained in an order designed to appear random to the processor ($J_0 \rightarrow J_2 \rightarrow J_{13} \rightarrow J_4 \rightarrow J_{10} \dots J_{15}$). Architecturally, the victim is programmed to jump directly from J_0 to the end of J_{15} . However, we flush the targets for the victim from the cache, causing it to speculatively execute the same chain as trained by the attacker until the targets are fetched from memory. At this point, the victim state is rolled back.

To determine whether a gadget J_i is executed by the victim, we instrument them with memory accesses (line 2) to unique addresses A_i . Using per-address Flush+Reload channels (see Section 2.2), we can determine which gadgets are executed: if gadget J_i is executed speculatively, the access to A_i is faster in the Reload phase. Knowing which gadgets were actually executed by the victim allows us to infer how many indirect jumps were successfully poisoned.

Figure 3a shows results from running this experiment on two generations of Intel processors, the i7-8700 and the i7-6700K, both with and without the latest microcode updates. On each machine, we use 10 sets of 1,000 runs, plotting the median of the fraction of times the n^{th} gadget was executed by the victim. The limits show the minimum and maximum fractions across the sets. On both processors, up to four gadgets can be chained with more than 50% success. However, the probabilities of chaining five or more gadgets drops drastically, with less than 10% success for reaching six gadgets. A preliminary investigation suggests that TLB misses are not responsible for this trend, as moving the gadgets or the cache-lines for the Flush+Reload channels to 2MB hugepages do not improve it. We also see that microcode updates do not significantly affect the median success rate.

4.1.2 Chaining gadgets through return instructions

We now study the chaining of gadgets terminated by return instructions (`ret`). Modern processors use the Return Stack Buffer (RSB) to predict the target of a `ret` instruction when the return address on the stack is not immediately available. Return addresses are pushed onto the RSB by `call` instructions, and are popped by `ret` instructions. Using unmatched function calls, attackers can push excess values onto the RSB

and cause misprediction on later returns.

To show the possibility of chaining gadgets using RSB we consider two experiments. In the first, the attacker and victim execute on the same thread. In the second, the attacker and victim run on different threads within the same process, using a `futex` to interleave their execution on a single core.

Same-thread chaining In this experiment, the attacker uses function calls to push a sequence of addresses onto the RSB. The victim executes subsequently, its return instructions using predictions from the RSB. The addresses on the RSB lead to a sequence of gadgets, each of which accesses memory at an unique address before executing a (poisoned) `ret` instruction. The memory accesses form a Flush+Reload channel to determine which of the gadgets were executed by the victim.

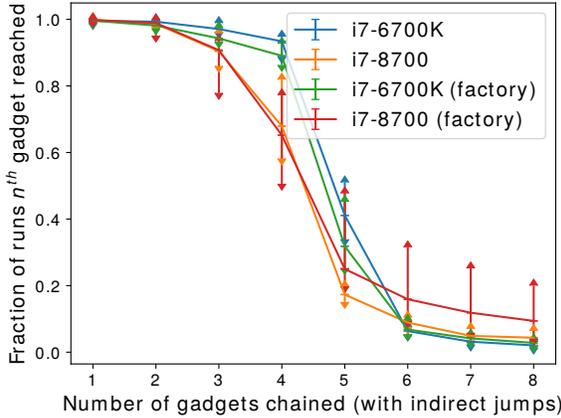
Figure 3b shows results of our experiments on an i7-6700K, an i7-8700 (16 RSB entries each), and a Xeon(R) E5-1620 (24 RSB entries). These experiments demonstrate the chaining of up to five gadgets with a reasonable success rate on the i7-6700K processor and up to two gadgets on the Xeon(R) E5-1620. The success rate drops precipitously to practically zero for more gadgets on all processors.

Cross-thread chaining In this experiment, the attacker poisons the RSB (as in the previous experiment) before using a `futex` to switch to the victim thread. Due to the limited size of the RSB, and its pollution during the context switch (there are multiple function calls within the kernel code) only a few RSB entries remain untouched for the victim. The victim’s execution is again similar to the previous experiment. On a i7-6700, the attacker can consistently chain up to two gadgets on the victim. On a Xeon(R) E5-1620, up to three gadgets can be chained using the RSB.

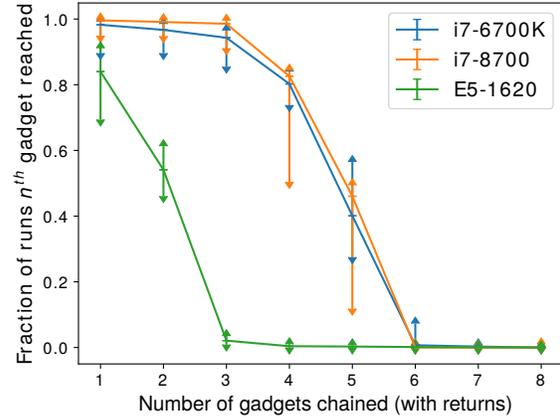
4.2 Proof-of-Concept

We now demonstrate the power of the SpecROP exploit technique. The proof-of-concept (PoC) attack is based on the SMOtherSpectre attack, which leaks specific bits using a side-channel based on port contention. Specifically, the SMOtherSpectre leakage gadget targets the least-significant bit (LSB) of the register `rdx`. In our PoC, we use this leakage gadget in different chains, augmenting the leakage capabilities of SMOtherSpectre. While this PoC uses the SpecROP approach to ameliorate one limitation of the SMOtherSpectre attack, we believe that it is indicative of the improvement possible for other attacks.

Our concept attack improves the SMOtherSpectre attack by leaking eight bits of the register, not just the LSB. The attacker achieves this by using shift gadgets for performing right-shift operations on register `rdx` before being redirected to the leakage gadget. Note that the attack can trivially be extended to leak the rest of the register. Figure 4 shows the flow of control on the victim. The attack starts at the basic block ending in



(a) Using indirect jumps



(b) Using ret instructions

Figure 3: The median success rate of chaining gadgets of different lengths on various processors. The limits represent the maximum and minimum rate across runs. Entries marked "factory" represents runs without microcode updates.

an indirect jump (labeled `jmp` in the figure). The actual jump target is the `end` block. By ensuring that the branch target is unavailable, the attacker causes the victim to start speculative execution, following the gadget chain trained by the attacker. The attacker repeats this process several times, leading the victim across the different paths to the leakage gadget. Along each path, the register holding the secret is shifted by different offsets, so that the leakage gadget ultimately leaks different bits of the key.

Let us illustrate the process of leaking different bits. At the jump gadget, suppose register `rdx` holds a 8-bit secret $s = \{s_i | i \in [0, 7]\}$ which the attacker wishes to leak. When the attacker directly chains the `jmp` to the leakage gadget, the value leaked is s_0 , the LSB of `rdx`. Instead, when the attacker chains `jmp` \rightarrow `shift 1` \rightarrow `leak`, the register `rdx` holds s_1 in the LSB at the leakage gadget. The leakage gadget now encodes s_1 into the side-channel. Similarly, on repetitions where the attacker chains `jmp` \rightarrow `shift j` \rightarrow `leak`, the bit s_j is leaked. Therefore, by progressively redirecting the victim's speculative control flow through different SpecROP chains, the attacker extends the leakage scope of SMOtherSpectre.

The SpecROP attack is not constrained to leaking individual bits. It is the characteristic of the leakage channel used in this POC that a bit is leaked at a time. With a cache-based side-channel which leaks a byte at a time, and *addition gadgets* which manipulate the pointer used to access secrets, a SpecROP attacker can leak an entire byte per iteration.

This proof-of-concept attack models a behavior which is typical of many programs, for example OpenSSL. C++ virtual function calls use indirect jumps, and hold a pointer to the object as the first parameter. For a virtual function call within a loop, the attacker may run different chains on different iterations. If the attacker can access different secrets by chaining different sequences of gadgets, the attacker may progressively

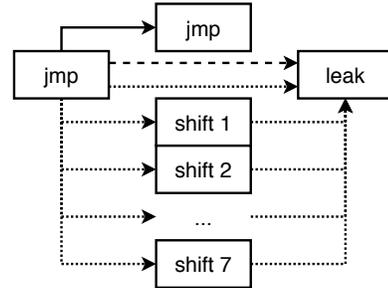


Figure 4: The flow of control during non-speculative execution (solid line), SMOtherSpectre attack (dashed line) and in SpecROP attack (dotted lines).

leak multiple secrets as the victim executes.

In our laboratory proof-of-concept, the attacker and victim run in separate processes. We run the experiment on an i7-6700K processor with microcode updates disabled. This allows cross-process branch poisoning on a shared physical core. With updated microcode, the attacker model changes to that described in [Section 4.1](#).

Our laboratory proof-of-concept was used to leak 1,024 randomly generated bytes. As discussed previously, different chains leak each of the 8 bits per byte, and we treat samples for each bit as a separate channel in the evaluation. For each channel, we collect 1,024 attacker SMOther timing samples (a measure of port contention with the victim), corresponding to 1,024 randomly generated "secret" bits on the victim. We then separate the attacker timings into two sets, depending on the value of corresponding secret bit on the victim, and plot the probability distribution function (pdf) for each set. When the distributions are clearly distinguishable, it means that the attacker's timing is strongly correlated to the victim's secret and can be classified with a high accuracy. [Figure 5](#)

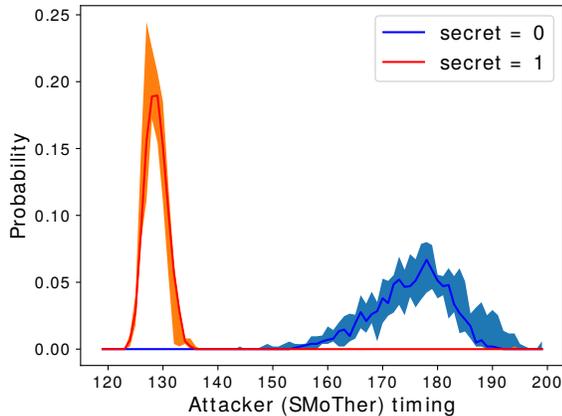


Figure 5: SMoTher timing for multiple gadget chain, separated according to the actual "secret" bit values. We have plotted the ranges of the individual probability distribution functions, separated by the secrets they represent, zero and one.

shows the results of the experiment. Our plot aggregates the results, showing the range of probabilities across the channels. The plot reveals that the attacker timings are similar across all channels, and that there is a clear separation within the distributions based on the actual secret. This means that the attacker can choose a threshold (around 140 cycles in this case), and classify each timing sample as a zero or one with a high accuracy. In fact, we can accurately guess victim secrets across all channels with an accuracy ranging between 99% and 100%.

In this experiment, the channel for bit 0 uses the shortest chain, directly connecting the `jmp` gadget to the leakage gadget. In fact, this chain is identical to the base SMoTherSpectre attack. This chain requires a single poisoned branch, whereas the chains for leaking other bits require two. We have seen (in Section 4.1.1) that longer chains lead to a diminishing probability of reaching the final (leakage) gadget. Therefore, the leakage accuracy for channel 0 is the target for the other channels. In fact, channel 0 leaks with 100% accuracy, and even the worst channel has an accuracy of greater than 99%. This shows that for a gadget chain of length two, SpecROP allows us to augment the leakage scope of SMoTherSpectre without suffering any loss of accuracy.

4.3 OpenSSL attack

In this section, we describe a realistic attack on a target program using OpenSSL's generic EnVelop (EVP) interface to encrypt/decrypt data. This attack improves upon the base SMoTherSpectre attack on the same target, enabling the attacker to leak an additional bit of the secret by modifying a pointer held in register `rdx` using an arithmetic gadget. By poisoning the BTB, the base SMoTherSpectre attack redirects

an indirect call in the `EVP_EncryptUpdate` function directly to a leakage gadget. The register `rdx` holds a pointer to the secret plaintext and this gadget leaks a bit of it from memory. In contrast, the SpecROP attacker first redirects the indirect call to an arithmetic gadget which increments `rdx` by a constant (for eg. `0x40`), and subsequently to a leakage gadget. As a result, the attacker leaks a different bit from the plaintext. With different arithmetic gadgets, this approach can vastly increase the leakage scope of SMoTherSpectre. Figure 6 illustrates this: the leftmost path shows the speculative control-flow in the base attack, and the other paths illustrate the leakage possible through different chains of gadgets. All the processing and leakage gadgets are taken from `glibc`, which is likely to be linked for most C programs, and are listed in full in Appendix B.

In particular, we implemented an attack using the chain which increments `rdx` by `0x40`. The basic procedure of the attack is very similar to SMoTherSpectre: the attacker and victim are threads in the same process running on logical cores on an SMT (hyperthreaded) physical core. Over 100,000 runs, the victim sets/resets the targeted bit in the plaintext and initiates an AES encryption. Concurrently, the attacker thread passes through a sequence of indirect jumps to poison the BTB. In contrast to basic SMoTherSpectre, this chain poisons more than one indirect branch on the victim. During the consequent period of speculative execution on the victim, the attacker times a sequence of instructions using `rdtsc` timestamps. Due to port contention, the attacker's readings should be correlated to the victim's secret. After the run, we separate the attacker's timings into sets based on the actual value of the victim's secret, and use the Student's t-test to validate that the distributions are actually distinguishable with high confidence (95%).

We ran our attack on an i7-8700 processor. Compared to SMoTherSpectre, the victim in our attack executed the leakage gadget in 33% of the runs instead of 80%. This is explained by our observations in Section 4.1.1: the success rate of reaching the final gadget drops with the length of the chain. Despite the added noise which results from runs where the leakage gadget was not reached, the Student's t-test reports that the distributions are distinguishable with 95% confidence. The test reports a timing difference of $4.41\% \pm 0.06\%$.

5 Specification: Gadget Search

To automate and improve the search for complex gadget chains, we develop a tool. The goal of our gadget search tool is two-fold:

1. To characterize the presence of processing gadgets which will enable an attacker to perform useful transformations on program state. Section 3.1.1 lists the target gadget classes.
2. To automate the process of finding gadget chains as per a set of constraints (described below).

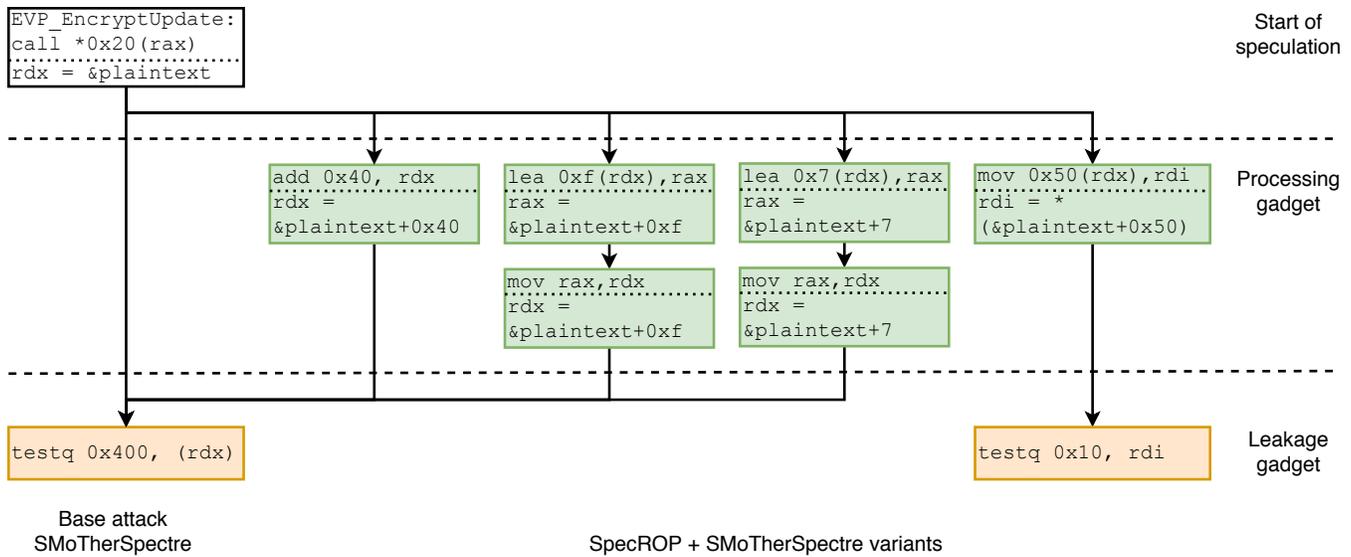


Figure 6: SpecROP allows an attacker to extend the leakage scope of the SMoTherSpectre attack on OpenSSL, targeting plaintext during encryption. The base attacker can leak a bit from byte 1 of the plaintext, whereas SpecROP chains enable leakage from bytes 8, 16, 65 and 80 using different processing gadgets. The relevant register state is shown below the dotted lines for each gadget.

The first goal allows us to support our claim that gadgets enhance the capability of an attacker to access and leak secrets. The second goal will enable attackers to construct useful chains from gadgets in real binaries, where the number of individual gadgets is large, and intractable for manual analysis.

Constraint handling SpecROP gadget chains have to respect constraints to prevent scenarios which stop speculation. For example, a gadget which loads an `rip` relative offset into register `rax` prevents speculation on a later indirect jump using the same register. Another constraint is that the register holding the secret must not be overwritten. A final constraint is that the gadget chain must make the secret available at the location (register/memory) disclosed by the leakage gadget. Other constraints are important for specific side-channels, for example a NetSpectre attacker requires intermediate gadgets to not use AVX instructions.

SpecFication uses symbolic representation of `x86_64` instructions to model their effects on processor state. This approach allows us to both compose the effects of instructions to express the effects of a gadget and to express constraints over our gadgets. As in ROP-chain tools [9], SpecFication starts by enumerating every address which can be interpreted as a valid instruction sequence ending in an indirect jump or return instruction. For each of these sequences, we model the semantics of the instructions over the registers. Currently, we only handle certain classes of instructions such as data movement, logic, arithmetic and branch instructions. SpecFication uses the Z3 theorem prover [10] for testing constraints over each sequence.

5.1 SpecFication architecture

SpecFication works in three phases: (i) binary disassembly and preprocessing, (ii) gadget characterization, and (iii) constraint enforcement.

In the *binary disassembly phase*, the Capstone [11] framework disassembles our target binaries. We create instruction sequences, including unintended instructions, which end in a return or indirect jump. Since we prioritize short gadgets, we limit the length of explored sequences to 6 instructions. We also remove gadgets containing specific instructions such as unintended control-flow and privileged instructions.

Based on the intermediate representation of the gadgets provided by Capstone, we map gadget semantics of the gadgets in the *characterization phase*. We express the semantics of each instruction in the Hoare logic space and compose the effects of all constituent instruction to generate the overall effect of the gadgets. This makes the gadgets amenable to processing by the Z3 theorem prover [10] in the *solving phase*. An alternate approach would be to leverage previous work which provide the formal specification of `x86_64` instructions, such as Strata [12] and Dasgupta et. al. [13]. For determining the effect of a gadget chain, the code for each of the individual gadgets must be composed after removing the terminating control-flow instructions (which the attacker will poison).

5.2 Evaluation and Results

We now evaluate the effectiveness of SpecFication in finding usable SpecROP gadgets in common libraries. Particularly,

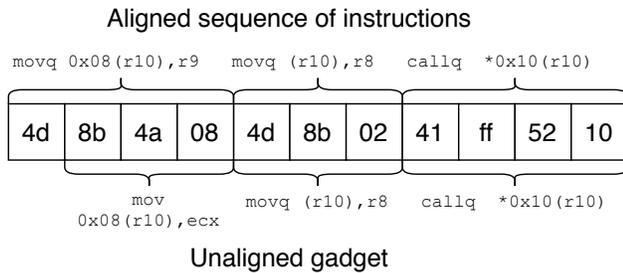


Figure 7: Example of an unaligned SpecROP gadget found in `libcrypto`

we look at instances of generic gadget types discussed in [Section 3.1.1](#).

Target libraries We have chosen a set of target libraries based on their ubiquity and security criticality. Specifically, we analyze:

- `libcrypto` from OpenSSL v1.1.1d,
- `mod_ssl`, `mod_proxy` and `mod_http2` from Apache v2.4.41,
- `libdl` v2.30, and
- `libc` v2.30.

Testing setup `SpecFication` is written in Python, and depends on `Capstone` v4.0 and `Z3` v4.8. All reported running times were measured on an i7-8700 processor with 16GB of memory running Debian 10 and Linux v5.4.

Results In the binary disassembly phase, `SpecFication` creates SpecROP gadgets: sequences of instructions ending in a return or indirect jump (an endpoint). For the analyzed binaries (which range from a few kilobytes to a few megabytes), the number of gadgets we analyze range from a few hundreds to thousands. [Table 2](#) highlights statistics about the number of gadgets processed by `SpecFication`. Note that there are roughly 10 gadgets for each unique endpoint in these binaries. A large fraction of the gadgets also contain at least one unintended instruction.

As a particular use case, we report statistics for a run in which we search for data movement gadgets which load from memory at any address with register `r12` as a base. We can see that the tool finds at least one usable gadget in each library, the exception being the smallest library (`libdl`). The running time for the tool ranges from a few seconds to a few minutes depending on the size of the binary. The constraint solving phase, which involves calling the `Z3` solver, is the largest contributor to the runtime.

We report statistics on the occurrence of gadgets classified as per the classes described in [Section 3.1.1](#).

- Arithmetic gadgets: [Table 4](#) highlights the number of arithmetic gadgets found in our target libraries, separated

as per the register on which the operation is done. There are a larger number of arithmetic gadgets operating on the first eight registers (`rax` to `rbp`) than on the remaining (`r8` to `r15`). We do find a large number of gadgets, specially in `libc` which operate on the first four argument registers used by the System-V calling convention: `rdi`, `rsi`, `rdx` and `rcx`. This enables a SpecROP attacker to perform ample range of computation with function arguments: for example, if these arguments are pointers to secrets, the attacker can manipulate and access different parts of the secret. Finally, some gadgets target registers `rsp` and `rbp` which allow the attacker to access secrets on the stack.

- Shift gadgets: We found a smaller number of gadgets performing bit movement on registers: 25 in `libcrypto`, 95 in `libc` and a handful in other libraries. A detailed breakdown of the occurrence of such gadgets is shown in [Appendix A](#).
- Data movement gadgets: We searched for gadgets in the target libraries which can move data between unique pairs of source and destination registers. Overall, there are a maximum of 240 unique pairs from the 16 general purpose registers (ignoring sub-registers) in `x86_64`. [Table 3](#) reports the number of such gadgets in each library, as well as the number of gadgets which result from unintended instructions (one in five gadgets, on average). By chaining more than one such gadget, we can increase the number of register pairs, allowing greater flexibility in data movement. The column labeled "Chained" shows the number of register pairs possible by chaining two data-movement gadgets. In line with the general aim of gadget chaining, most libraries exhibit a significant increase in data-movement possibilities with increased chain length. In fact, chaining two gadgets allows 84% of the register pairs possible with chains of any length.
- Leakage gadgets: [Table 4](#) also reports (in even rows) the occurrence of gadget leaking information into cache-based side channels, assuming that loads to secret-dependent addresses can leak information.

The results in this section not only illustrate the abundance of usable SpecROP gadgets in real libraries, but also demonstrate the practicality of using binary analysis for performing automated gadget search with formally specifiable constraints. This methodology has allowed us to construct practical SpecROP chains against OpenSSL ([Figure 2](#)) and similar to Spectre ([Listing 1](#)). We have demonstrated that this methodology can streamline the often manual process of finding gadgets in new binaries, and for newer side-channel attacks.

6 Mitigation

The mitigations against a SpecROP attack include generic defenses against SEA, such as preventing speculation, pre-

Library	Binary size	Endpoints	Gadgets	Unaligned	Data-movement gadgets addressing r12		
					Endpoints	Gadgets	Analysis time (s)
libcrypto	3.3M	1,209	13,437	9,545	19	65	233
libc	1.8M	1,282	15,044	10,130	5	13	333
libdl	15K	22	266	205	0	0	4
mod_ssl	235K	48	490	332	2	4	8
mod_proxy	131K	30	338	246	1	1	5
mod_http2	244K	112	1,113	796	3	8	18

Table 2: Number of endpoints, gadgets, and unaligned gadgets found per library. We also show statistics for a particular use-case: searching for gadgets which load from an address based on register r12

Library	Data movement gadgets (unaligned)	Chained	Analysis time (s)
libcrypto	116 (9)	210	5,644
libc	101 (23)	204	8,432
libdl	2 (2)	2	305
mod_ssl	32 (10)	47	419
mod_proxy	34 (11)	46	295
mod_http2	27 (5)	72	875

Table 3: Occurrence of data movement gadgets moving data between registers. We report how many unique combinations of source and destination x86_64 registers were found in each library.

venting branch predictor poisoning and control flow integrity. Other defenses particular to SpecROP might include limits on the number of branches followed speculatively. Static binary analysis techniques are inherently limited in their ability to detect whether usable SpecROP chains exist in binaries due to the large number of possible targets for a poisoned indirect jump or return instruction, and the exponential explosion in the number of possible sequences with the number of chained gadgets.

Preventing speculation in software The simplest protection against SEA is to restrict speculation following sensitive branches (where there is access to secrets). This can be done by using serializing instructions (for example `cuserid`), or memory fences (for example `lfence`) when the side-channel uses load instructions. If implemented by a shotgun approach, the performance implications are significant. However, we have shown how SpecROP chains expand the reach of SEA to access secrets, precluding fine-grained application of speculation barriers. Another mitigation, *retpolines* [14], protects indirect jumps by replacing them with return instructions. It also pollutes the Return Stack Buffer with the address of an infinite loop to prevent speculation on the `ret`. A practical, though partial, mitigation would be to identify code which might access secrets (e.g., array accesses following a bounds

check), and insert *retpolines* on subsequent indirect calls and returns. This approach would still be vulnerable to gadget chains where the attacker is able to manipulate existing state to access secret state in unforeseen, and therefore unprotected, gadgets.

Limiting speculation in hardware Architectural proposals which limit the number of speculative control-flow instructions will effectively constrain the maximum length of a SpecROP chain, reducing the attack surface. However, the typical speculation window on a modern, high-performance processor extends to hundreds of instructions, where it is likely to contain tens of speculative control-flow instructions. We have already seen that it is impractical to chain more than 4-5 gadgets on these processors [Figure 3a](#). A smaller limit on speculative control-flow instructions (say 2-3) may have an unacceptable performance overhead.

Preventing leakage Numerous proposals exist for mitigating SEA by closing the leakage channels, particularly for memory-based channels. *InvisiSpec* [15] proposes a separate buffer to hold speculatively loaded values, preventing them from affecting cache state. *DAWG* [16] dynamically partitions the cache to prevent cross-context cache channels. Other proposals which restrict execution of instructions dependent on speculatively accessed values [17] will effectively close all speculative side-channels, even if they do not prevent the chaining of gadgets.

Preventing branch poisoning Existing processors offer some degree of protection against control-flow hijacking across processes, and between different processor privilege levels (particularly userspace and the kernel) [18–20], either in hardware or through microcode updates. However, as we can see from results in [Table 1](#), these measures do not completely mitigate all branch poisoning attack scenarios.

Enforcing control-flow integrity Any control-flow integrity mechanism aiming to mitigate SpecROP must account for speculative control-flow. Therefore, off-the-shelf

Library	Type	rax	rbx	rcx	rdx	rdi	rsi	rsp	rbp	r8	r9	r10	r11	r12	r13	r14	r15
libcrypto	A	665	259	34	78	69	65	186	48	0	0	0	0	15	33	10	3
	L	218	255	137	192	238	59	899	159	26	21	7	0	128	106	106	35
libc	A	889	317	128	171	419	421	32	3	13	29	2	0	8	12	6	15
	L	188	171	65	96	570	110	643	231	8	8	36	360	34	28	41	32
libdl	A	25	6	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	L	9	11	0	8	0	0	43	0	0	0	0	0	0	0	0	0
mod_ssl	A	12	8	0	4	0	0	0	0	0	0	0	0	0	0	2	0
	L	2	38	0	2	0	10	10	1	0	0	0	0	8	2	23	0
mod_proxy	A	12	6	0	0	2	0	1	0	0	0	0	0	7	2	2	0
	L	8	2	0	0	0	0	13	0	0	0	0	0	2	0	0	1
mod_http2	A	46	5	0	5	0	9	0	0	0	0	0	0	0	0	0	0
	L	32	48	6	22	43	15	112	60	0	0	0	0	8	13	16	46

Table 4: Occurrence of arithmetic (A) and leakage (L) gadgets, listed on the first and second rows respectively for each library

CFI mechanisms are insufficient. Hardware mechanisms include Intel’s upcoming CET technology [21] which plans to limit speculative execution following a jump or call. However, as we see with the example in Figure 2, SpecROP attacks are still possible within the limits imposed by the early implementations¹. Since the set of targets allowed by CET is a superset of the actual set of targets, it remains to be seen if this imprecision can be used for speculative exploitation. There are newer proposals [22] for more complete CFI under speculative execution.

7 Related Work

The hypothesis that an advanced SEA might chain multiple code gadgets by having “multiple outstanding speculative changes of address stream caused by branch prediction” first appeared in a white-paper by ARM [23]. The paper, however, lacks an investigation of this idea, its practicality or the existence of the required code-gadgets. A similar hypothesis also appears in a more recent work by Canella et al. [24].

More recently, Mambretti et al. [25] demonstrated a practical SEA using multiple, chained gadgets to leak information following a mispredicted conditional branch. Unlike SpecROP, the attack requires a memory corruption bug in the victim binary to be able to inject arbitrary return addresses on to the stack. The targets for the return instructions used to chain gadgets comes from the overwritten stack. In contrast, SpecROP considers a stronger attacker model and is more stealthy. Our attacker cannot write arbitrary values to the victim’s stack, and leaves no architecturally visible traces.

Bulck et al. [26] mention the possibility of using Load Value Injection(LVI) to transiently poison the values loaded from memory and used by return and indirect jump instruc-

tions, thereby chaining gadgets like in ROP/JOP. LVI depends on faulty behavior in specific CPU models. Moreover, their attack requires the ability to induce faults in the victim on the load instructions which they wish to poison. In general, this makes their attack practical only against victims running within an SGX enclave. The paper also does not comment on the practicality of causing multiple LVIs.

Other work which uses automated program analysis include Spectector [27] and oo7 [28]. Both of these works aim to prevent Spectre-like attacks by doing a static analysis of the code. oo7 statically applies taint analysis to binaries to check whether tainted values (secrets) can affect the outcome of conditional branches and speculative memory accesses. Spectector analyzes binaries for speculative code paths which leave microarchitectural traces which the architecturally intended path does not. Given the exact path through a SpecROP chain, Speculator would be able to detect that the instruction sequence is leaky. However, neither of these analyses can practically detect information leakage resulting from a SpecROP gadget chain since the set of available sequences to analyze grows exponentially with the length of the gadget chain.

SplitSpectre [29] also proposes an approach to implement the Spectre attack where the target does not contain any single gadget which loads the secret and performs a dependent load, instead relying on an attacker with the ability to inject the second load on the natural control-flow following the first. In contrast, SpecROP reuses existing gadgets for both loads. Further, Spectector will be able to detect that the code sequence used in SplitSpectre is leaky since there is a direct path between the dependent loads.

8 Conclusion

Through SpecROP, we extend our understanding of practical Speculative Execution Attacks by studying the ability to

¹The early implementations of Intel CET will restrict execution following indirect jumps/calls to 8 instructions and 5 loads.

chain multiple gadgets. We demonstrate that poisoning multiple control flow instructions (returns and indirect jumps) is possible on modern processors. In fact, on tested processors, we can poison up to 4 indirect jumps with more than 50% success rate. This opens up the potential for attackers to use gadget chains instead of monolithic gadgets. With the use of the SpecROP technique, we show how generic gadgets can extend the reach of secrets accessible by an attacker, with an example of how this methodology may be applied to access and leak the AES key from a pointer to the context. We also demonstrate a practical attack which is able to leak multiple plaintext bits from a victim during encryption using the OpenSSL library.

To facilitate the gadget search, we design SpecFication, a gadget search tool for searching for generic SpecROP gadgets, from which we build up useful chains. Applying SpecFication to existing code bases, we see the abundance of small, generic SpecROP gadgets. We also study the possible mitigation techniques for SpecROP attacks. Our results lead us to believe that modern processors and programs are indeed vulnerable to a SpecROP attack, and that processors require hardware solutions for preventing malicious influence on branch/return prediction.

We have published as open-source the code for our experiments, the proof-of-concepts and SpecFication to enable others to further explore this methodology. The code is available at the GitHub repository <https://github.com/HexHive/specrop-public>.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868).

References

- [1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
- [2] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 552–561. [Online]. Available: <https://doi.org/10.1145/1315245.1315313>
- [3] V. van der Veen, D. Andriess, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later," in *CCS*, Oct. 2017. [Online]. Available: [Paper=http://vvdveen.com/publications/newton.pdf](http://vvdveen.com/publications/newton.pdf)
[Web=https://www.vusec.net/projects/newton](https://www.vusec.net/projects/newton)
- [4] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, Eds. ACM, 2011, pp. 30–40. [Online]. Available: <https://doi.org/10.1145/1966913.1966919>
- [5] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, C. Rossow and Y. Younan, Eds. USENIX Association, 2018. [Online]. Available: <https://www.usenix.org/conference/woot18/presentation/koruyeh>
- [6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 785–800. [Online]. Available: <https://doi.org/10.1145/3319535.3363194>
- [7] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 719–732. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>
- [8] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., vol. 11735. Springer, 2019, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-030-29959-0_14
- [9] S. Schirra, "Ropper - rop gadget finder and binary information tool," <http://scoding.de/ropper/>, 2014.

- [10] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [11] N. A. Quynh, “Capstone: Next-gen disassembly framework,” *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.
- [12] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, “Stratified synthesis: Automatically learning the x86-64 instruction set,” in *Programming Language Design and Implementation (PLDI)*. ACM, June 2016. [Online]. Available: <http://dx.doi.org/10.1145/2908080.2908121>
- [13] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, “A complete formal semantics of x86-64 user-level instruction set architecture,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1133–1148. [Online]. Available: <https://doi.org/10.1145/3314221.3314601>
- [14] P. Turner, “Retpoline: a software construct for preventing branch-target-injection,” <https://support.google.com/faqs/answer/7625886>, 2018.
- [15] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum),” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, p. 1076. [Online]. Available: <https://doi.org/10.1145/3352460.3361129>
- [16] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, “DAWG: A defense against cache timing attacks in speculative execution processors,” in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 974–987. [Online]. Available: <https://doi.org/10.1109/MICRO.2018.00083>
- [17] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “NDA: preventing speculative execution attacks at their source,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 572–586. [Online]. Available: <https://doi.org/10.1145/3352460.3358306>
- [18] Intel Corporation, “Deep dive: Indirect branch restricted speculation,” <https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation>, accessed: 2020-03.
- [19] —, “Deep dive: Indirect branch predictor barrier,” <https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier>, accessed: 2020-03.
- [20] —, “Deep dive: Single thread indirect branch predictors,” <https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors>, accessed: 2020-03.
- [21] —, “Control-flow enforcement technology specification,” <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>, May 2019, accessed: 2020-03.
- [22] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, “SPECCFI: mitigating spectre attacks using CFI informed speculation,” *CoRR*, vol. abs/1906.01345, 2019. [Online]. Available: <http://arxiv.org/abs/1906.01345>
- [23] R. Grisenthwaite, “Cache speculation side-channels,” January 2018.
- [24] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtvushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 249–266. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/canella>
- [25] A. Mambretti, A. Sandulescu, A. Sorniotti, W. K. Robertson, E. Kirda, and A. Kurmus, “Bypassing memory safety mechanisms through speculative control flow hijacks,” *CoRR*, vol. abs/2003.05503, 2020. [Online]. Available: <https://arxiv.org/abs/2003.05503>
- [26] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, “Lvi: Hijacking transient execution through microarchitectural load value injection,” in *41th IEEE Symposium on Security and Privacy (S&P’20)*, 2020.
- [27] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sanchez, “Spectector: Principled detection of speculative information flows,” in *IEEE Symposium on Security and Privacy*. IEEE, May 2020. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/spectector-principled-detection-of-speculative-information-flows/>

- [28] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, “oo7: Low-overhead defense against spectre attacks via program analysis,” *IEEE Transactions on Software Engineering*, 2019.
- [29] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. K. Robertson, and A. Kurmus, “Speculator: a tool to analyze speculative execution attacks and mitigations,” in *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, D. Balenson, Ed. ACM, 2019, pp. 747–761. [Online]. Available: <https://doi.org/10.1145/3359789.3359837>

A SpecROP: Shift gadgets

Table 5 breaks down the shift gadgets found in each library based on the register which is operated on.

B OpenSSL attack gadgets

This section fully lists the gadgets used in the OpenSSL attacks shown in Figure 2 and Figure 6. The gadgets are found in `glibc` and `libcrypto`.

B.1 Processing gadgets

The following processing gadget increments the register `rdi` by a constant `0x20`. It is found in `libcrypto`.

```
1f6cc6: lea    0x20(rdi),rdi
1f6cca: callq  *0x18(%rax)
```

The following processing gadget loads a pointer from the memory referenced by `rdi` at offset `0x58` into register `rax`. It is found in `libcrypto`.

```
b2f1b: mov    0x58(%rdi),%rax
b2f1f: retq
```

The following processing gadget increments the value in `rdx` by `0x40`. It is found in `libcrypto`.

```
16b87f: add    0x40, rdx
16b883: add    rdx, rsi
16b886: add    rdx, rdi
16b889: lea   0x2e920(rip), r11
16b890: movsxd (r11, rdx, 4), rcx
16b894: add    r11, rcx
16b897: jmp    *rcx
```

The following processing gadget stores the value of `rdx + 0xf` in `rax`. It is found in `glibc`.

```
17df7e: lea    0xf(rdx),rax
17df82: retq
```

The following processing gadget stores the value of `rdx + 7` in `rax`. It is found in `glibc`.

```
17df26: lea    0x7(rdx),rax
17df2a: retq
```

The following processing gadget stores the value of `rax` in `rdx`. It is found in `glibc`.

```
12afdf: mov    rax,rdx
12afe2: callq  *0x28(r12)
```

The following processing gadget loads 8 bytes at address referenced by `rdx` at an offset of `0x50` into register `rdi`. It is found in `glibc`.

```
12ef33: mov    0x50(rdx),rdi
12ef37: mov    rdx,rsi
12ef3a: callq  *rax
```

B.2 Leakage gadgets

The following gadget leaks the LSB of register `rax`. It is found in `glibc`.

```
cf6ac: mov    -0x1b0(rbp),rdx
cf6b3: mov    -0x1a8(rbp),rdi
cf6ba: mov    r15d,esi
cf6bd: or     0x2,esi
cf6c0: mov    rbx,rcx
...
cf939: test   0x1,al
cf93b: je     cf6ac
cf941: mov    r15d,r13d
cf944: movb   0x0,(rdx)
cf947: mov    -0x1b0(rbp),rdx
cf94e: and    0xfffff7ef,r13d
cf955: mov    rbx,rcx
cf958: mov    r13d,esi
cf95b: or     0x2,esi
...
```

The following gadget leaks the 3rd LSB from the byte at offset 1 from the pointer in `rdx`. It is found in `glibc`.

```
f5393: testq  0x400,(rdx)
f539a: je     f5382
f539c: mov    -0xb0(rbp),rdi
f53a3: mov    -0xf0(rbp),edx
f53a9: mov    (rdi,rax,8),rax
f53ad: test   edx,edx
f53af: mov    rax,0x50(rbx)
...
f5382: add    0x1,rax
f5386: add    0x20,rdx
f538a: cmp    rax,-0x100(rbp)
...
```

Library	rax	rbx	rcx	rdx	rdi	rsi	rsp	rbp	r8	r9	r10	r11	r12	r13	r14	r15
libcrypto	3	10	2	6	3	1	0	0	0	0	0	0	0	0	0	0
libc	21	44	0	29	1	0	0	0	0	0	0	0	1	1	0	0
libdl	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
mod_ssl	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mod_proxy	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mod_http2	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5: Occurrence of shift gadgets in each library, broken down based on the affected register

The following gadget leaks the 6th LSB from byte referenced by the pointer in `rbp`. The first instruction is an unaligned instruction. It is found in `glibc`.

```
6aa23: testb 0x20, (rbp)
6aa27: je 6aa2f
6aa29: mov (rbx), rax
6aa2c: orl $0x20, (rax)
6aa2f: add $0x18, rsp
6aa33: mov r13, rdi
6aa36: pop rbx
6aa37: pop rbp
6aa38: pop r12
6aa3a: pop r13
...
```

The following gadget leaks the 5th LSB from the register `rdi`. It is found in `glibc`.

```
8ed34: test 0x10, rdi
8ed3b: je 8ed5a
8ed3d: movdqu (rdi, rsi, 1), xmm0
8ed42: pcmpeqb (rdi), xmm0
8ed46: pmovmskb xmm0, edx
8ed4a: sub 0xffff, edx
8ed50: jne 8ee80
8ed56: add 0x10, rdi
8ed5a: mov r11, r10
8ed5d: and 0xfffffffffffffe0, r10
```

Never Trust Your Victim: Weaponizing Vulnerabilities in Security Scanners

Andrea Valenza
University of Genova
andrea.valenza@dibris.unige.it

Gabriele Costa
IMT School for Advanced Studies Lucca
gabriele.costa@imtlucca.it

Alessandro Armando
University of Genova
alessandro.armando@unige.it

Abstract

The first step of every attack is reconnaissance, i.e., to acquire information about the target. A common belief is that there is almost no risk in scanning a target from a remote location. In this paper we falsify this belief by showing that scanners are exposed to the same risks as their targets. Our methodology is based on a novel attacker model where the scan author becomes the victim of a counter-strike. We developed a working prototype, called RevOK, and we applied it to 78 scanning systems. Out of them, 36 were found vulnerable to XSS. Remarkably, RevOK also found a severe vulnerability in Metasploit Pro, a mainstream penetration testing tool.

1 Introduction

Performing a network scan of a target system is a surprisingly frequent operation. There can be several agents behind a scan, e.g., attackers that gather technical information, penetration testers searching for vulnerabilities, Internet users checking a suspicious address. Often, when the motivations of the scan author are unknown, it is perceived by the target as a hostile operation. However, scanning is so frequent that it is largely tolerated by the target. Even from the perspective of the scanning agent, starting a scan seems not risky. Although not completely stealthy, an attacker can be reasonably sure to remain anonymous by adopting basic precautions, such as proxies, virtual private networks and onion routing.

Yet, expecting an acquiescent scan target is a mere assumption. The scanning system may receive poisoned responses aiming to trigger vulnerabilities in the scanning host. Since most scanning systems generate an HTML report, scan authors can be exposed to attacks via their browser. This occurs when the scanning system permits an unsanitized flow of information from the response to the user browser. To illustrate, consider the following, minimal HTTP response.

```
HTTP/1.1 200 OK
Server: nginx/1.17.0
...
```

A naive scanning system might extract the value of the `Server` field (namely, the string `nginx/1.17.0` in the above example) and include it in the HTML report. This implicitly allows the scan target to access the scan author's browser and inject malicious payloads.

In this paper we investigate this attack scenario. We start by defining an attacker model that precisely characterizes the threats informally introduced above. To the best of our knowledge, this is the first time that such an attacker model is defined in literature. Inspired by the attacker model, we define an effective methodology to discover cross-site scripting (XSS) vulnerabilities in the scanning systems and we implement a working prototype. We applied our prototype to 78 real-world scanning systems. The results confirm our expectation: several (36) scanning systems convey attacks. All of these vulnerabilities have been notified through a responsible disclosure process.

The most remarkable outcome of our activity is possibly an XSS vulnerability enabling remote code execution (RCE) in Rapid7 Metasploit Pro. We show that the attack leads to the complete takeover of the scanning host. Our notification prompted Rapid7 to undertake a wider assessment of their products based on our attacker model.

The main contributions of this paper are:

1. a novel attacker model affecting scanning systems;
2. a testing methodology for finding vulnerabilities in scanning systems;
3. RevOK, a prototype implementation of our testing methodology;
4. an analysis of the experimental results on 78 real-world scanning systems, and;
5. three application scenarios highlighting the impact of our attacker model.

This paper is structured as follows. Section 2 recalls some preliminary notions. Section 3 presents our attacker model.

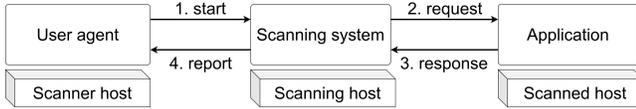


Figure 1: Abstract architecture of a scanning system.

We introduce our methodology in Section 4. Our prototype and experimental results are given in Section 5. Then, we present the three use cases in Section 6, while we survey on the related literature in Section 7. Finally, Section 8 concludes the paper.

2 Background

In this section we recall some preliminary notions necessary to correctly understand our methodology.

2.1 Scanning systems

A scanning system is a piece of software that (i) stimulates a target through network requests, (ii) collects the responses, and (iii) compiles a report. Security analysts often use scanning systems for technical information gathering [8]. Scanning systems used for this purpose are called *security scanners*. Our definition encompasses a wide range of systems, from complex vulnerability scanners to simple ping utilities.

Figure 1 shows the key actors involved in a scan process. Human analysts use a user agent, e.g., a web browser, to select a target, possibly setting some parameters, and start the scan (1. start). Then, the scanning system crafts and sends request messages to the target (2. request). The scanning system parses the received response messages (3. response), extracts the relevant information and provides the analyst with the scan result (4. report). Finally, the analysts inspect the report via their user agent.

Whenever a scanning system runs on a separate, remote scanning host, we say that it is provided *as-a-service*. Instead, when the scanner and scanning hosts coincide, we say that the scanning system is *on-premise*.

A popular, command line scanning system is Nmap [25]. To start a scan, the analyst runs a command from the command line, such as

```
nmap -sV 172.16.1.26 -oX report.xml
```

Then, Nmap scans the target (172.16.1.26) with requests aimed at identifying its active services (-sV). By default, Nmap sends requests to 1,000 frequently used TCP ports and collects responses from the services running on the target. The result of the scan is then saved (-oX) on `report.xml`. Interestingly, some web applications, e.g., Nmap Online [15], provide the functionalities of Nmap as-a-service.

Scanning systems are often components of larger, more complex systems, sometimes providing a browser-based GUI.

For instance, Rapid7 Metasploit Pro is a full-fledged penetration testing software. Among its many functionalities, Metasploit Pro also performs automated information gathering, even including vulnerability scanning. The reporting system of Metasploit Pro is based on an interactive Web UI used to browse the report.

2.2 Taint analysis

Taint analysis [26] refers to the techniques used to detect how the information flows within a program. Programs read inputs from some sources, e.g., files, and write outputs to some destinations, e.g., network connections. For instance, taint analysis is used to understand whether an attacker can force a program to generate undesired/illegal outputs by manipulating some of its inputs. A *tainted flow* occurs when (part of) the input provided by the attacker is included in the (tainted) output of the program. In this way, the attacker controls the tainted output which can be used to inject malicious payloads to the output recipient.

2.3 Cross-site scripting

Cross-site scripting (XSS) is a major attack vector for the web, stably in the OWASP Top 10 vulnerabilities [12] since its initial release in 2003. Briefly, an XSS attack occurs when the attacker injects a third-party web page with an executable script, e.g., a JavaScript fragment. The script is then executed by the victim’s browser. The simplest payload for showing that a web application suffers from an XSS vulnerability is

```
<script>alert(1)</script>
```

that causes the browser to display an alert window. This payload is often used as a proof-of-concept (PoC) to safely prove the existence of an XSS vulnerability.

There are several variants to XSS. Among them, *stored* XSS has highly disruptive potential. An attacker can exploit a stored XSS on a vulnerable web application to permanently save the malicious payload on the server. In this way, the attack is directly conveyed by the server that delivers the injected web page to all of its clients. Another variant is *blind* XSS, in which the attacker cannot observe the injected page. For this reason, blind XSS relies on a few payloads, each adapting to multiple HTML contexts. These payloads are called *polyglots*. A remarkable example is the polyglot presented in [11] which adapts to at least 26 different contexts.

3 Attacker model

The idea behind our attacker model is sketched in Figure 2 (bottom), where we compare it with a traditional web security attacker model (top). Typically, attackers use a security scanner to gather technical information about a target application. If the application suffers from some vulnerabilities,

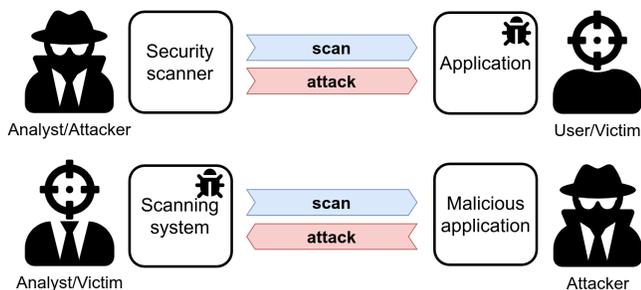


Figure 2: Comparison between attacker models.

attackers can exploit them to deliver an attack towards their victims, e.g., the application users. On the contrary, in our attacker model attackers use malicious applications to attack the author of a scan, e.g., a security analyst.

Here are the two novelties of our attacker model.

1. Attacks are delivered through HTTP *responses* instead of *requests*.
2. Attackers exploit the vulnerabilities of scanning systems to strike their victims, i.e., the scan initiator.

Below, we detail the attacker’s goal and capabilities.

Attacker goal. The objective of the attacker is to directly strike the analyst. To do so, the attacker exploits the vulnerabilities of the target scanning system and its reporting system to hit the analyst user agent. In this work, we assume that the user agent is a web browser. This assumption covers every as-a-service scanning system, as well as many on-premise ones, which generate HTML reports. As a consequence, here we focus on XSS which is a major attack vector for web browsers. As usual in XSS, the attacker succeeds when the victim’s browser executes a piece of attacker-provided code, e.g., JavaScript.

Attacker capabilities. First, we state that the attacker has adequate resources to detect vulnerabilities in scanning systems before deploying the malicious application. However, the attacker capabilities do not include the possibility of observing the internal logic of the scanning system. That is, our attacker operates in black-box mode.

Secondly, our attacker has complete control over the malicious application, e.g., the attacker owns the scanned host. However, we do not assume that the attacker can force the victim to initiate the scanning process.

4 Testing methodology

In this section, we define a vulnerability detection methodology based on our attacker model.

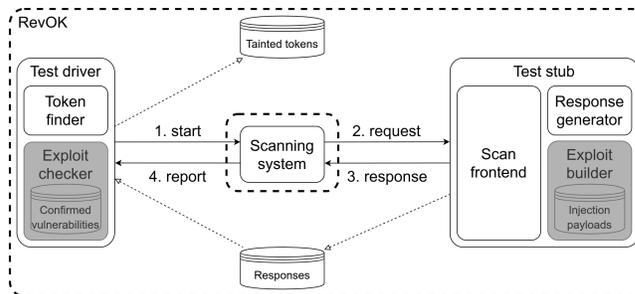


Figure 3: Phase 1 – find tainted flows.

4.1 Test execution environment

Our methodology relies on a *test execution environment* (TEE) to automatically detect vulnerabilities in scanning systems. In particular, a *test driver* simulates the user agent of the security analyst, while a *test stub* simulates the scanned application. Our TEE can (i) start a new scan, (ii) receives the requests of the scanning system, (iii) craft the responses of the target application, and (iv) access the report of the scanning system. Intuitively, the TEE replicates the configuration of Figure 1. In this configuration, the test driver is executed by the scanner host, and the test stub runs on the scanned host. In general, the test driver is customized for each scanning system under testing. For instance, it may consist of a Selenium-enabled [14] browser stimulating the web UI of the scanning system.

Both the test driver and the test stub consist of some sub-modules. These submodules are responsible for implementing the two phases described below.

4.2 Phase 1: tainted flows enumeration

The first phase aims at detecting the existing tainted destinations in the report generated by the scanning system. Having a characterization of the tainted flows is crucial to deal with the input transformation logic of the target scanning system. In general, since payloads may be arbitrarily modified before being displayed in the report, detecting actual injections is non-trivial. Instead, through this phase, injections can be detected just by monitoring tainted destinations. The process is depicted in Figure 3. Initially, the test driver asks the scanning system to perform a scan of the test stub. The scan logic is not exposed by the scanning system and, thus, it is opaque from our perspective. Nevertheless, it generates some requests toward the test stub. Each request is received by the *scan frontend* and dispatched to the *response generator*, which crafts the response.

The response generation process requires special attention. One might think that a single, general-purpose response is sufficient. However, some scanning systems process the responses in non-trivial ways. For instance, they may abort the scan if a malformed or suspicious response is received. For this reason, we proceed as follows. First, we generate a

response template, i.e., an HTTP response containing variables, denoted by t . Response templates are generated from a fuzzer through a *probabilistic context-free grammar* (PCFG). A PCFG is a tuple (N, Σ, R, S, P) , where $G = (N, \Sigma, R, S)$ is a context-free grammar such that N is the set of non-terminal symbols, Σ is the set of terminal symbols, R are the production rules and S is the starting symbol. The additional component of the PCFG, namely $P : R \rightarrow [0, 1]$, associates each rule in R with a probability, i.e., the probability to be selected by the fuzzer generating a string of G . Additionally, we require that P is a probability distribution over each non-terminal α , in symbols

$$\forall \alpha \in N. \sum_{(\alpha \mapsto \beta) \in R} P(\alpha \mapsto \beta) = 1$$

In the following, we write $\alpha \mapsto_p \beta$ for $P(\alpha \mapsto \beta) = p$ and $\alpha \mapsto_{p_1} \beta_1 |_{p_2} \dots |_{p_n} \beta_n$ for $\alpha \mapsto_{p_1} \beta_1, \dots, \alpha \mapsto_{p_n} \beta_n, \alpha \mapsto_{p_e} ""$ (where "" is the empty string).

The probability values appearing in our PCFG are assigned according to the results presented in [20, 21]. There, the authors provide a statistical analysis of the frequency of real response headers as well as a list of information-revealing ones. Such headers are thus likely to be reported by a scanning system. Finally, when the frequency of a field is not given (e.g., for variables), we apply the uniform distribution.

An excerpt of our PCFG is given in Figure 4. For the sake of presentation, here we omit some of the rules and we refer the interested reader to the project web site¹. The grammar defines the structure of a generic HTTP response (Resp) made of a version (Vers), a status (Stat), a list of headers (Head), and a body (Body). Variables t are all fresh and they can appear in several parts of the generated response template. In particular, variables can be located in status messages (i.e., Succ , Redr , ClEr and SvEr), header fields (i.e., Serv , PwBy , Locn , SetC , CntT , AspV , MvcV , Varn , StTS , CnSP , XSSP and FrOp) and body. For instance, a field can be $\text{Server: nginx}/t$, where $\text{nginx}/$ is a server type (SrvT , omitted for brevity).

The response template is then populated by replacing each variable with a *token*. A token is a unique sequence of characters that is both *recognizable*, i.e., it has a negligible probability of appearing by chance, and *uninterpreted*, i.e., the browser treats it as plain text, when appearing in an HTML document. All tokens are mapped to the responses containing them. Responses are stored in a database. Finally, the test driver matches the tokens appearing in the responses database with those occurring in the scan report. Such tokens are evidence that there are tainted flows in the internal logic of the scanning system. Tokens mark the source and the sink of a flow in the response and report, respectively. All these tokens are stored in the tainted tokens database.

```

Resp  $\mapsto_1$  Vers Stat Head Body
Vers  $\mapsto_{0.5}$  "HTTP/1.0" | $_{0.5}$  "HTTP/1.1"
Stat  $\mapsto_{0.554}$  Succ | $_{0.427}$  Redr | $_{0.013}$  ClEr | $_{0.006}$  SvEr
Succ  $\mapsto_{0.5}$  "200 OK" | $_{0.5}$  "200"  $t$ 
Redr  $\mapsto_{0.386}$  "301 Moved Permanently" | $_{0.386}$  "301"  $t$ 
      | $_{0.114}$  "302 Found" | $_{0.114}$  "302"  $t$ 
ClEr  $\mapsto_{0.26}$  "403 Forbidden" | $_{0.26}$  "403"  $t$ 
      | $_{0.24}$  "404 Not Found" | $_{0.24}$  "404"  $t$ 
SvEr  $\mapsto_{0.5}$  "500 Internal Server Error" | $_{0.5}$  "500"  $t$ 
Head  $\mapsto_1$  Serv PwBy Locn SetC CntT AspV MvcV Varn
       $\hookrightarrow$  StTS CnSP XSSP FrOp
Serv  $\mapsto_{0.475}$  "Server:"  $t$  | $_{0.475}$  "Server:" SrvT  $t$ 
PwBy  $\mapsto_{0.24}$  "X-Powered-By: php"
      | $_{0.24}$  "X-Powered-By:"  $t$ 
Locn  $\mapsto_{0.315}$  "Location:" Link | $_{0.315}$  "Location:"  $t$ 
Link  $\mapsto_{0.516}$  "https://"  $t$ 
      | $_{0.167}$  "http://"  $t$  ":"8899"
      | $_{0.135}$  "http://"  $t$  ":"8090"
      | $_{0.065}$  "http://"  $t$  "/login.lp"
      | $_{0.059}$  "/nocookies.html"
      | $_{0.058}$  "cookiechecker?uri="/
SetC  $\mapsto_{0.175}$  "Set-Cookie:" Ckie
Ckie  $\mapsto_{0.471}$  "__cfduid="  $t$  | $_{0.394}$  "PHPSESSID="  $t$ 
      | $_{0.087}$  "ASP.NET Session="  $t$ 
      | $_{0.048}$  "JSESSIONID="  $t$ 
CntT  $\mapsto_{0.07}$  "X-Content-Type-Options: nosniff"
      | $_{0.07}$  "X-Content-Type-Options:"  $t$ 
AspV  $\mapsto_{0.5}$  "X-AspNet-Version:"  $t$ 
MvcV  $\mapsto_{0.5}$  "X-AspNetMvc-Version:"  $t$ 
Varn  $\mapsto_{0.5}$  "X-Varnish:"  $t$ 
StTS  $\mapsto_{0.5}$  "Strict-Transport-Security:" STSA
STSA  $\mapsto_{0.111}$  "max-age=" N+
      | $_{0.111}$  "max-age="  $t$ 
      | $_{0.111}$  "max-age=" N+ "; preload"
      | $_{0.111}$  "max-age="  $t$  "; preload"

```

Figure 4: Response template grammar (excerpt).

4.3 Phase 2: vulnerable flows identification

The second phase aims to confirm which tainted flows are actually vulnerable. We use PoC exploits to confirm the vulnerability. The workflow is depicted in Figure 5. As for the first phase, the test driver launches a scan of the test stub. When the test stub receives the requests, the exploit builder extracts a response from the responses database. Then, the response is injected with a PoC exploit. More precisely, a tainted token is selected among those generated during Phase 1. The tainted token in the response is replaced with a payload taken from a predefined injection payload database. In general, a vulnerability is confirmed by the test driver according to predefined, exploit-dependent heuristics. Although tainted flows can be subject to different types of vulnerabilities, as discussed in Section 3, we focus on XSS. Thus, the heuristics implemented by the exploit checker consists of recognizing a vulnerable flow when an alert window is spawned by the corresponding, tainted flow. Finally, the exploit checker stores the vulnerable flows in the confirmed vulnerabilities database.

The definition of injection payload is non-trivial. Since our

¹<https://github.com/AvalZ/RevOK>

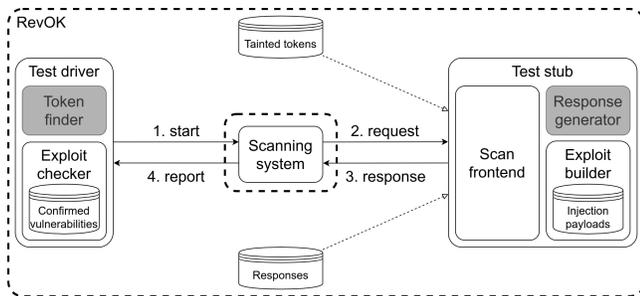


Figure 5: Phase 2 – find vulnerable flows.

TEE applies to both on-premise and as-a-service scanning systems, some issues must be considered. The first issue is testing performances. As a matter of fact, scanning systems can take a considerable amount of time to perform a single scan. Moreover, as-a-service scanning systems should not be flooded by requests to avoid degradation of the quality of service. For these reasons, we aim to limit the number of payloads to check.

As discussed in Section 2.3, polyglots allow us to test multiple contexts with a single payload. In this way, we increase the success probability of each payload and, thus, we reduce the overall number of tests.

In principle, we might resort to the polyglot of [11], which escapes 26 contexts. However, its length (144 characters) is not adequate since many scanning systems shorten long strings when compiling their reports, so preventing the exploit from taking place. To avoid this issue, we opted for polyglots such as `"' />".` This is rendered by the browser when appearing inside both an HTML tag and an HTML attribute. The reason is that the initial `"` and `'` allow the payload to escape from quoted attributes.

Furthermore, delivering the JavaScript payload in `onerror` has two advantages. First, it circumvents basic input filtering methods, e.g., blacklisting of the `script` string. Secondly, our payload applies to both static and dynamic reports. More precisely, a static report consists of HTML pages that are created by the scanning system and subsequently loaded by the analyst's browser. Instead, a dynamic report is loaded by the browser and updated by the scanning system during the scan process. The HTML5 standard specification [16, §8.4.3] clearly states that browsers can skip the execution of dynamically loaded scripts. For this reason, our payload binds the script execution to an `error` event that we trigger using a broken image link (i.e., `src='x'`). A concrete example of this scenario is discussed in Section 6.2.

5 Implementation and results

In this section, we present our prototype RevOK. We used it to carry out an experimental assessment that we discuss in

Section 5.3.

5.1 RevOK

Our prototype consists of two modules: the test driver and the test stub. We detail them below.

Test driver A dedicated test driver is used for each scanning system. The test driver (i) triggers a scan against the test stub, (ii) saves the report in HTML format and (iii) processes the report to detect tainted and vulnerable flows (in Phase 1 and 2, respectively). While (iii) is the same for all the scanning systems, (i) and (ii) may vary.

In general, the implementation of (i) and (ii) belongs to two categories depending on whether the scanning system has a programmable interface or only a GUI. When a programmable interface is available, we implement a Python 3 client application. For instance, we use the native `os` Python module to launch Nmap so that its report is saved in a specific location (as describe in Section 2). Similarly, we use the `requests` Python library² to invoke the REST APIs provided by a scanning system and save the returned HTML report. Instead, when the scanning system only supports GUI-based interactions, we resort to GUI automation. In particular, we use the Selenium Python library³ for browser-based GUIs and `PyAutoGUI`⁴ for desktop GUIs. In the case of GUI automation, the test driver repeats a sequence of operations recorded during a manual test.

Finally, for the report processing step (iii) we distinguish between two operations. The tainted flow detection trivially searches the report for the injected tokens provided by the response generator (see below). Instead, vulnerable flows are confirmed by checking the presence of alert windows through the Selenium function `switch_to_alert()`.

Test stub For the response generator, we implemented the PCFG grammar fuzzer detailed in Section 4.2 in Python. Tokens are represented by randomly-generated Universally Unique Identifiers [22] (UUID). A UUID consists of 32 hexadecimal characters organized in 5 groups that are separated by the `-` symbol. An example UUID is `018d54ae-b0d3-4e89-aa32-6f5106e00683`. As required in Section 4.2, UUIDs are both recognizable (as collisions are extremely unlikely to happen) and uninterpreted (as they contain no HTML special characters).

On the other hand, starting from a response, the exploit builder replaces a given UUID with an injection payload. Payloads are taken for a predefined list of selected polyglots, as discussed in Section 4.3.

²<https://requests.readthedocs.io>

³<https://selenium-python.readthedocs.io/>

⁴<https://pyautogui.readthedocs.io>

5.2 Selection criteria

We applied our prototype implementation to 78 scanning systems. The full list of scanning systems, together with our experimental results (see Section 5.3), is given in Table 1. There, we use \odot and \oplus to distinguish between as-a-service and on-premise scanning systems, respectively.

For our experiments, we searched for scanning systems included in several categories. In particular, we considered security scanners, server fingerprinting tools, search engine optimization (SEO) tools, redirect checkers, and more. From these, we removed scanning systems belonging to the following categories.

- Abandonware, i.e., on-premise scanning systems that were not maintained in the last 5 years.
- Paywalled, i.e., scanning systems that are not free and have no trial version.
- Scheduled, i.e., as-a-service scanning systems that only perform periodic scans, not controlled by the analyst.

5.3 Results

We applied RevOK to the scanning systems of Table 1. For each scanning system, we used RevOK to execute 10 scan rounds (see Section 4) and we listed all the detected tainted and vulnerable flows. As a result, we discovered that 67 scanning systems have tainted flows and, among them, 36 are vulnerable to XSS.

In Table 1, for each scanning system we report the number of tainted and vulnerable flows (T and V, respectively) detected by RevOK. After running RevOK, we also conducted a manual vulnerability assessment of each scanning system. The assessment consisted of a review of each tainted flow, followed by a manual payload generation (see below).

Under column M, \checkmark indicates that an XSS vulnerability was found by a human analyst starting from the outcome of RevOK. It is worth noticing that only in one case, i.e., DupliChecker, RevOK resulted in a false negative w.r.t. the manual analysis. By investigating the causes, we discovered that DupliChecker performs URL encoding on the tainted locations. This encoding, among other operations, replaces white spaces with %20, thus invalidating our payloads. To effectively bypass URL encoding, we replaced white spaces (U+0020) with non-breaking spaces (U+00A0) that are not modified. Thus, we defined a new polyglot payload that uses non-breaking spaces and we added it to the injection list included in RevOK. Using this new payload, RevOK could also detect the vulnerability in DupliChecker.

At the time of writing, all the vulnerabilities detected by RevOK have been reported to the tool vendors and are undergoing a responsible disclosure process (see Appendix A).

In Figure 6 we show the frequency of the tainted and vulnerable flows over the 14 fields considered by RevOK. Location

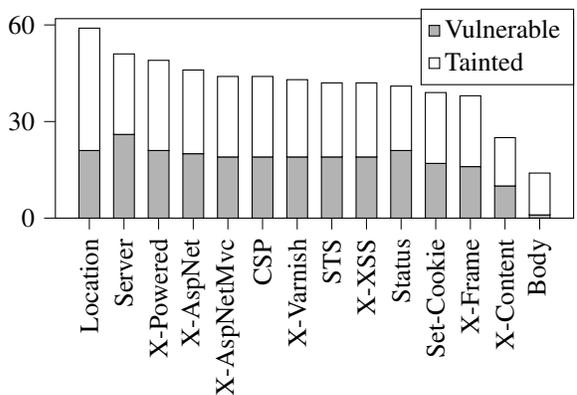


Figure 6: Frequency of tainted and vulnerable flows.

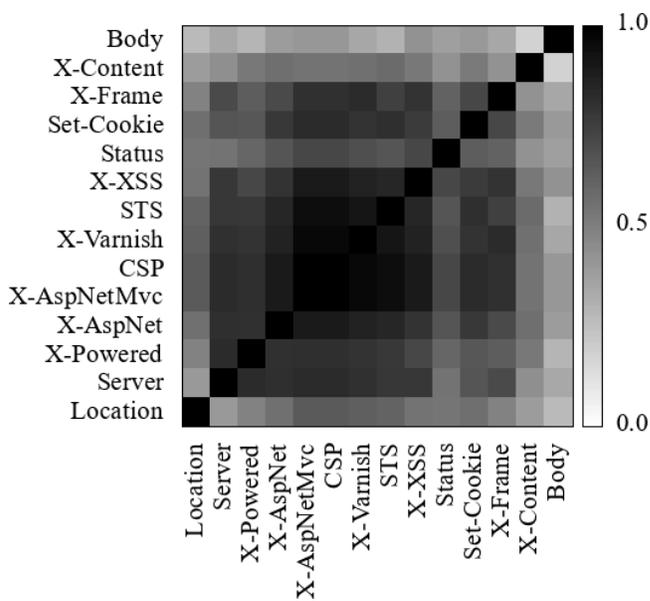


Figure 7: Correlation between tainted fields.

has 59 tainted flows, the highest number, and 21 vulnerable flows. Server only has 51 tainted flows, but it has 26 vulnerable flows, the highest number. On the other hand, Body has only 14 tainted flows and only 1 vulnerable flow. This highlights that most scanning systems sanitize the Body field in their reports. The reason is that HTTP responses most likely contain HTML code in their Body. Thus, sanitization is mandatory to preserve the report layout. Also, the Body field is often omitted by the considered scanning systems.

In Figure 7 and Figure 8 we show the correlation matrices for tainted and vulnerable fields, respectively. From these matrices we observe a few, relevant facts. We briefly discuss them below.

The first observation is that the Body field is almost unrelated to the other fields, both in terms of tainted and vulnerable flows. This is somehow expected since the Body field is often neglected as discussed above.

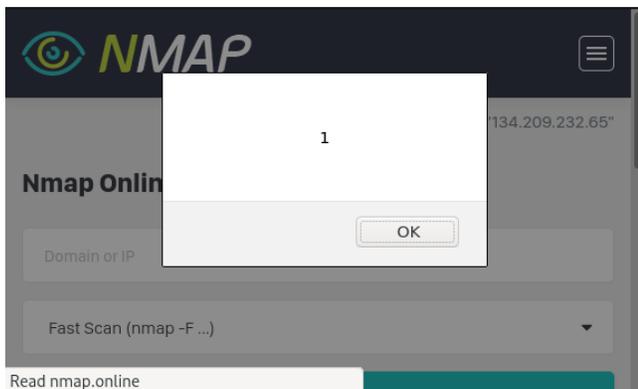


Figure 9: XSS PoC on Nmap Online.

its impact if an attacker were to use it in the wild. For each subclass of scanning systems, we chose a representative that we present as a concrete case study: Nmap Online for as-a-service scanning systems, Metasploit Pro for on-premise ones, and CheckShortURL [5] for redirect checkers.

6.1 Scan Attribution

Attack attribution is a hot topic since it is often difficult or even impossible to achieve. The main reasons are the structure of the network and some state-of-the-art technologies that enable clients anonymity. For instance, analysts can use proxies, virtual private networks, and onion routing to hide the actual source of the requests from the recipient. However, an injected browser may be forced to send identifying data directly inside the HTTP requests, so making network-level anonymization techniques ineffective. In this section, we show how to attribute scans using our attacker model through an application scenario based on Nmap Online [15].

Nmap Online vulnerability Nmap Online is a web application providing some of the functionalities of Nmap. Users can scan a target with Nmap without having to install it on their machine. Furthermore, since requests originate from the Nmap Online server, users can stay anonymous w.r.t. the scan target. When users start a scan, they select the target IP and the scan type. The Nmap Online website scans its target and displays the retrieved information to the user, e.g., server type and version.

Nmap Online reports suffered from an XSS vulnerability.⁵ Figure 9 shows an injected report. The injection occurs on the Server response header. In this case, the Server field was set to `<script>alert(1)</script>`.

Browser hooking Since there is no guarantee that more than one scan will occur, we recur to browser hooking, which

can be obtained with a single XSS payload. A hooked browser becomes the client in a command and control (C2) infrastructure, thus actively querying the C2 server for instructions. This allows the attacker to submit arbitrary commands afterward even when no other scans occur.

An effective way to achieve browser hooking is through BeEF [2]. In particular, the BeEF C2 client is injected via the script `hook.js`. For instance, we can deploy `hook.js` by setting the Server header to `<script src='http://[C2]/hook.js'></script>` where [C2] is the IP address of the C2 server.

Fingerprinting The BeEF framework includes modules⁶ for fingerprinting the victim host. For instance, the `browser` module allows us to get the browser name, version, visited domains, and even starting a video streaming from the webcam. Similarly, the `host` module allows us to retrieve data such as physical location and operating system details. Some of these operations, e.g., browser fingerprinting, require no victim interaction. Instead, others need the victim to take some actions, e.g., explicitly grant permission to use the webcam. To overcome these hurdles, attackers usually employ auxiliary techniques, e.g., credential theft, implemented by some BeEF modules, e.g., *social engineering*. Finally, the overall fingerprinting process can be automated through the BeEF *autorun rule engine* [1].

6.2 Scanning host takeover

On-premise scanning systems, which run on the analyst's host, may have privileged, unrestricted access to the underlying platform. In some cases, on-premise systems are provided with a user interface that includes both the reporting system and a control panel. When such a user interface is browser-based, a malicious scan target can inject commands in the reporting system and perform lateral movements by triggering the scanning system controls.

The attack strategy abstractly described above must be implemented through concrete steps that are specific to the scanning system. In this section, we show an implementation of this attack strategy for the popular scanning system Metasploit Pro. In particular, we show how to perform lateral movements leading to a complete scanning host takeover through remote code execution (RCE). Finally, we carry out an impact evaluation.

CVE-2020-7354 and CVE-2020-7355 Metasploit Pro is a full-fledged penetration testing framework. It has a browser-based UI that integrates both a scan reporting system and many controls for running the most common tasks, including host scanning. Each command is executed by the Metasploit back-end, which is stimulated through a REST API.

⁵The vulnerability was fixed on March 24, 2020.

⁶<https://github.com/beefproject/beef/wiki/BeEF-modules>

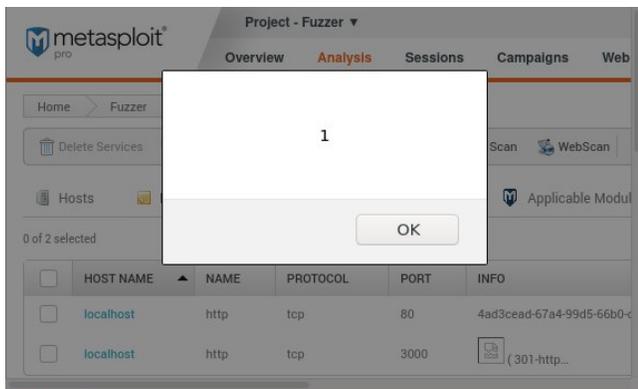


Figure 10: Stored XSS PoC on Metasploit Pro.

The vulnerability we found affects versions 4.17.0 and below. It was remediated on May 14, 2020, with patch 4.17.1.⁷ A malicious scan target can inject a Stored XSS payload in the UI. Multiple pages are vulnerable, e.g., `/hosts/:id` and `/workspaces/:id/services`.

Metasploit Pro fetches the Server header and displays it inside the *INFO* column with no sanitization. Figure 10 shows the effect of setting the Server header to `` (as described in Section 4.3).

Remote code execution We use the XSS vulnerability described above to gain a foothold in the browser on the scanning host. For instance, we can inject a BeEF hook to remotely interact with the browser (as in Section 6.1). The hooked browser is the steppingstone to interact with the Metasploit Pro UI and trigger its controls. Interestingly, Metasploit Pro includes a *diagnostic console*, i.e., an embedded terminal that allows the analyst to run arbitrary commands on the underlying operating system.⁸ Although the diagnostic console is disabled by default, the attacker can activate it through BeEF. In particular, the hooked browser is forced to perform a POST HTTP request to `/settings/update_profile` with the parameter `allow_console_access=1`. Since the diagnostic console is a browser-embedded Metasploit terminal emulator, the attacker can submit commands from the BeEF interface.

Takeover impact The Metasploit Pro documentation⁹ clearly states that “Metasploit Pro Users Run as Root. If you log in to the Metasploit Pro Web UI, you can effectively run any command on the host machine as root”. This opens a wide range of opportunities for the attacker. Among them,

⁷<https://help.rapid7.com/metasploit/release-notes/archive/2020/05/#20200514>

⁸<https://www.exploit-db.com/exploits/40415>

⁹<https://metasploit.help.rapid7.com/docs/metasploit-web-interface-overview>

the most impactful is to establish a *reverse shell*. The reasons are twofold. First, opening a shell on the scanning host allows the attacker to execute commands directly on the operating system of the victim. Thus, attacks are no longer tunneled through the initial vulnerability, which might become unavailable, e.g., if Metasploit Pro is terminated. Second, a reverse shell works well even when certain network facilities, such as firewalls and NATs, are in place. Indeed, although these facilities may prevent incoming connections, usually they allow outgoing ones. Once a reverse shell is established, the attacker can access a permanent, privileged shell on the victim host.

6.3 Enhanced phishing

The goal of a phishing attack is to induce the victim to commit a dangerous action, e.g., clicking an untrusted URL or opening an attachment. In this section, we show how our attacker model changes phishing attacks, using CheckShortURL as an application scenario.

Traditional Phishing A common phishing scenario is that of an unsolicited email with a link pointing to a malicious web page, e.g., `http://ev.il`. The phishing site mimics a reputable, trusted web page. For instance, the attacker may clone a bank’s web site so that unaware users submit their access credentials. Another technique is to provoke a reaction to an emotion, such as fear. This happens, for instance, with menacing alerts about imminent account locking and malware infections. Again, if victims believe that urgent action must be taken, they could overlook common precautions and, e.g., download dangerous files.

Defense mechanisms Most of the examples given above require the victim to open a phishing URL. Common, unskilled users typically evaluate the trustworthiness of a URL by applying their common sense.¹⁰ Nevertheless, techniques such as URL shortening and open redirects [27] masquerade the phishing URL to resemble a trusted domain.

Some online services may help the user to detect phishing attacks. For instance, reputation systems and black/white lists, e.g., Web of Trust¹¹, can be queried for a suspect URL. However, phishing URLs often point to temporary websites that are unknown to these systems.

Since browsers automatically redirect without asking for confirmation, in [27] the authors highlight that victims can defend themselves by checking where the URL redirects without browsing it. To this aim, several online services, e.g., CheckShortURL, do redirect checking to establish the final destination of a redirect chain. Typically, the chain is printed in a report that the user inspects before deciding whether to proceed or not.

¹⁰E.g., see <https://phishingquiz.withgoogle.com/>

¹¹<https://www.mywot.com>

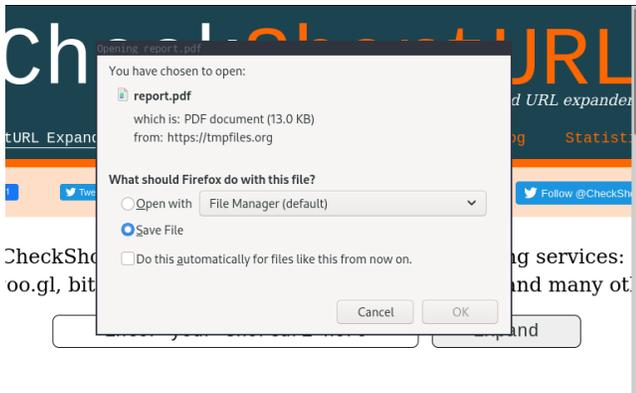


Figure 11: Phishing through CheckShortURL.

Exploiting redirect checkers Redirect locations are contained in the Location header of the HTTP response asking for a redirection. According to our attacker model, this value is controller by the attacker. Thus, if the victim uses a vulnerable redirect checker, the report may convey an attack to the user browser. Since the goal is phishing, the attacker has two possibilities, i.e., forcing the URL redirection and exploit the scanning system reputation.

In the first case, the attacker delivers an XSS payload such as `window.location = "http://ev.il/"`. When it is executed, the browser is forced to open the given location and to redirect the user to the phishing site.

The second case is even more subtle. Since the XSS attack is delivered by the scanning system, the attacker can perform a phishing operation and ascribe it to the reporting system. For instance, the attacker can make the user browser download a malicious file pretending to be the scanning system pdf report. In this way, the attacker abuses the reputation of the scanning system to lure the victim. This can be achieved with the following payload.

```
window.location="http://tmpfiles.org/report.pdf"
```

The effect of injecting such a payload in CheckShortURL is shown in Figure 11.

7 Related Work

In this section, we survey the related literature.

7.1 Attacking the attacker

Although not frequent in the literature, the idea of attacking the attackers is not completely new. Its common interpretation is that the victim of an attack carries out a counter-strike against the host of the aggressor. However, even tracking an attack to its actual source is almost impossible if the attacker takes proper precautions (as discussed in Section 6.1). To the best of our knowledge, we are the first to consider the response-based exploitation of the attackers scanning systems.

Djanali et al. [9] define a low-interaction honeypot that simulates vulnerabilities to lure the attackers to open a malicious website. When this happens, the malicious website delivers a browser exploitation kit. The exploitation relies on a LikeJacking [29] attack to obtain information about the attacker’s social media profile. Unlike our approach, their proposal substantially relies on social engineering and does not consider vulnerabilities in the attacker’s equipment.

Also, Sintov [28] relies on a honeypot to implement a *reverse penetration* process. In particular, his honeypot attempts to collect data such as the IP address and the user agent of the attacker. Again, this proposal amounts to retaliating against the attackers after identifying them.

In terms of vulnerabilities, some researchers already reported weaknesses in scanning systems. The closest to our work is CVE-2019-5624 [4], a vulnerability in RubyZip that also affects Metasploit Pro. This vulnerability allows attackers to exploit *path traversal* to create a *cron job* that runs arbitrary code, e.g., to create a reverse shell. To achieve this, the attacker must import a malicious file in Metasploit Pro as a new project. However, as for [9], this attack requires social engineering as well as other conditions (e.g., about the OS used by the attacker). As far as we know, this is the only other RCE vulnerability reported for Metasploit Pro. Instead, apart from ours, no XSS vulnerabilities have been reported.

7.2 Security scanners assessment

Several authors considered the assessment of security scanners. However, they mainly focus on their effectiveness and efficiency in detecting vulnerabilities.

Doupé et al. [10] present WackoPicko, an intentionally vulnerable web application designed to benchmark the effectiveness of security scanners. The authors provide a comparison of how open source and commercial scanners perform on the different vulnerabilities contained in WackoPicko.

Holm et al. [18] perform a quantitative evaluation of the accuracy of security scanners in detecting vulnerabilities. Moreover, Holm [17] evaluated the performance of network security scanners, and the effectiveness of remediation guidelines.

Mburano et al. [23] compare the performance of OWASP ZAP and Arachni. Their tests are performed against the OWASP Benchmark Project [13] and the Web Application Vulnerability Security Evaluation Project (WAVSEP) [6]. Both these projects aim to evaluate the accuracy, coverage, and speed of vulnerability scanners.

To the best of our knowledge, there are no proposals about the security assessment of scanning systems. Among the papers listed above, none consider our attacker model or, in general, the existence of security vulnerabilities in security scanners.

7.3 Vulnerability detection

Many authors proposed techniques to detect software vulnerabilities. In principle, some of these proposals can be applied to scanning systems.

The general structure of vulnerability testing environments was defined by Kals et al. [19]. Our TEE implements their abstract framework by adapting it to inject responses instead of requests. The main difference is our test stub, that receives the requests from the scanning system under test. We substitute the crawling phase with tainted flow enumeration (see Section 4.2). During the attack phase, we substitute the payload list with a list of polyglots, which reduces testing time. Our exploit checker implements their analysis module as we also deal with XSS.

Many authors have proposed techniques to perform vulnerability detection through dynamic taint analysis. For instance, Xu et al. [32] propose an approach that dynamically monitors sensitive sinks in PHP code. It rewrites PHP source code, injecting functions that monitor data flows and detect injection attempts.

Avancini and Ceccato [3] also use dynamic taint analysis to carry out vulnerability detection in PHP applications. Briefly, they implement a testing methodology aiming at maximizing the code coverage. To check whether a certain piece of code was executed, they rewrite part of the application under test to deploy local checks.

These approaches rely on inspecting and manipulating the source code of the application under test. Instead, we work under a black-box assumption.

Besides vulnerability detection, some authors even use dynamic taint analysis to implement exploit detection and prevention methodologies. Vogt et al. [30] prevent XSS attacks by combining dynamic and static taint analysis in a hybrid approach. Similarly, Wang et al. [31] detect DOM-XSS attacks using dynamic taint analysis. Both these approaches identify sensitive data sinks in the application code and monitor whether untrusted, user-provided input reaches them.

Dynamic taint analysis techniques were also proposed for detecting vulnerabilities in binary code.

Newsome and Song [24] propose *TaintCheck*, a methodology that leverages dynamic taint analysis to find attacks in commodity software. *TaintCheck* tracks tainted sinks and detects when an attack reaches them. It requires a monitoring infrastructure to achieve this.

Clause et al. [7] propose a generic dynamic taint analysis framework. Similarly to [24], Clause et al. implement their technique for x86 binary executables. However, the theoretical framework could be adapted to fit our methodology.

In principle, the exploit prevention techniques mentioned above might be used to mitigate some of the vulnerabilities detected by RevOK. However, they do not deal with vulnerability detection. Moreover, they require access to the application code.

8 Conclusion

In this paper we introduced a new methodology, based on a novel attacker model, to detect vulnerabilities in scanning systems. We implemented our methodology and we applied our prototype RevOK to 78 real-world scanning systems. Our experiments resulted in the discovery of 36 new vulnerabilities. These results confirm the effectiveness of our methodology and the relevance of our attacker model.

Acknowledgements. This paper was partially funded by EU H2020 research project SPARTA (grant agreement n.830892).

References

- [1] Wade Alcorn. Beef autorun rule engine. <https://github.com/beefproject/beef/wiki/Autorun-Rule-Engine>, Accessed March 19, 2020.
- [2] Wade Alcorn. *The Browser Exploitation Framework*, Accessed March 3, 2020.
- [3] Andrea Avancini and Mariano Ceccato. Towards Security Testing with Taint Analysis and Genetic Algorithms. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*, 2010.
- [4] Luca Carettoni. On insecure zip handling, Rubyzip and Metasploit RCE (CVE-2019-5624). <https://blog.doyensec.com/2019/04/24/rubyzip-bug.html>, Accessed March 19, 2020.
- [5] CheckShortURL. *CheckShortURL*, Accessed March 23, 2020.
- [6] Shay Chen. The Web Application Vulnerability Scanner Evaluation Project. <https://sourceforge.net/projects/wavsep/>, Accessed March 19, 2020.
- [7] James Clause, Wanchun Li, and Alessandro Orso. Dy-tan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.
- [8] MITRE Corporation. ATT&CK - Technical Information Gathering. <https://attack.mitre.org/tactics/TA0015/>, Accessed March 20, 2020.
- [9] Supeno Djanali, FX Arunanto, Baskoro Adi Pratomo, Abdurrazak Baihaqi, Hudan Studiawan, and Ary Mazharuddin Shiddiqi. Aggressive web application honeypot for exposing attacker's identity. In *Proceedings of the 1st International Conference on Information Technology, Computer, and Electrical Engineering*, 2014.

- [10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010.
- [11] Ahmed Elsobky. Unleashing an Ultimate XSS Polyglot. <https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot>, Accessed March 19, 2020.
- [12] OWASP Foundation. OWASP Top Ten. <https://owasp.org/www-project-top-ten/>, 2017.
- [13] OWASP Foundation. OWASP Benchmark Project. <https://owasp.org/www-project-benchmark/>, Accessed March 19, 2020.
- [14] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*, 2015.
- [15] MUNSIRADO Group. *Nmap Online*, Accessed March 3, 2020.
- [16] Web Hypertext Application Technology Working Group. *HTML Living Standard*, Last updated March 27, 2020.
- [17] Hannes Holm. Performance of automated network vulnerability scanning at remediating security issues. *Computers & Security*, 2012.
- [18] Hannes Holm, Teodor Sommestad, Jonas Almroth, and Mats Persson. A quantitative evaluation of vulnerability scanning. *Information Management & Computer Security*, 2011.
- [19] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *Proceedings of the 15th International Conference on World Wide Web*, 2006.
- [20] Arturs Lavrenovs and F Jesús Rubio Melón. HTTP Security Headers Analysis of Top One Million Websites. In *Proceedings of the 10th International Conference on Cyber Conflict (CyCon)*, 2018.
- [21] Arturs Lavrenovs and Gabor Visky. Investigating HTTP response headers for the classification of devices on the Internet. In *Proceedings of the 7th IEEE Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2019.
- [22] Paul Leach, Michael Mealling, and Rich Salz. A universally unique identifier (UUID) urn namespace. 2005.
- [23] Balume Mburano and Weisheng Si. Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark. In *Proceedings of the 26th International Conference on Systems Engineering (ICSEng)*, 2018.
- [24] James Newsome, Dawn Song, James Newsome, and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*, 2005.
- [25] Nmap project. *Nmap*, Accessed March 23, 2020.
- [26] 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*, 2010.
- [27] Craig A Shue, Andrew J Kalafut, and Minaxi Gupta. Exploitable Redirects on the Web: Identification, Prevalence, and Defense. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, 2008.
- [28] Alexey Sintsov. Honey-pot that can bite: Reverse penetration. In *Black Hat Europe Conference*, 2013.
- [29] SOPHOSLABS. Facebook worm: Likejacking. <https://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/>, Accessed on March 19, 2020.
- [30] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.
- [31] Ran Wang, Guangquan Xu, Xianjiao Zeng, Xiaohong Li, and Zhiyong Feng. TT-XSS: A novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting. *Journal of Parallel and Distributed Computing*, 2018.
- [32] Wei Xu, Sandeep Bhatkar, and R Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. *Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook*, 2005.

A Vulnerability Disclosure

All the vulnerabilities reported in this paper were promptly notified to the scanning system vendors. We based our responsible disclosure process on the ISO 29147¹² guidelines. Below, we describe each disclosure step in detail and the vendors feedback.

A.1 First contact

The first step of our responsible disclosure process consisted of a non-technical email notification to each vendor. We report our email template below.

```
Dear <scanning system vendor>,

my name is <identification and links>

As part of my research activity on a novel threat model, I found that your platform is most likely vulnerable to XSS attacks.
In particular, the vulnerability I discovered might expose your end-users to concrete risks.

For these reasons, I am contacting you to start a responsible disclosure process. In this respect, I am kindly asking you to point me to the right channel (e.g., an official bug bounty program or a security officer to contact).

Kind regards
```

We sent the email through official channels, e.g., contact mail or form, when available. For all the others, we tried with a list of 13 frequent email addresses, including security@, webmaster@, contact@, info@, admin@, support@.

In 5 cases the previous attempts failed. Thus, we submitted the corresponding vulnerabilities to OpenBugBounty.¹³

A.2 Technical disclosure

After the vendor answered our initial notification, providing us with the technical point of contact, we sent a technical report describing the vulnerability. The report was structured according to the following template, which was accompanied by a screenshot of the PoC exploit inside their system.

```
The issue is a Cross-Site Scripting
```

¹²<https://www.iso.org/standard/72311.html>

¹³<https://www.openbugbounty.org>

```
attack on your online vulnerability
scanning tool <scanning system name>.
```

This exposes your users to attacks, possibly leading to data leakage and account takeover.

A malicious server can answer with XSS payloads instead of its standard headers. For example, it could answer with this (minimal) HTTP response:

```
<minimal PoC for the scanning system>
```

Since your website displays this data in a report, this code displays a popup on the user page, but an attacker can include any JavaScript code in it, taking control of the user browser (see <https://beefproject.com/>), and hence make them perform actions on your website or steal personal information.

I attached a screenshot of the PoC running on your page. The PoC is completely harmless, both for your website and for you to test. I also hosted a malicious (but harmless) server here if you want to reproduce the issue: <test stub network address>

You can perform any scan you want against it (please let me know if it is offline).

In a few cases we extended the report with additional details, requested by some vendors. For example, some of them asked for the CVSSv3¹⁴ calculation link and an impact evaluation specifically referring their scanning system.

A.3 Vendors feedback

Out of the 36 notifications, we received 12 responses to the first contact message. All the responses arrived within 2 days. Among the notified vendors 5 fixed the vulnerability within 10 days. Another vendor informed us that, although they patched their scanning system, they started a more general investigation of the vulnerability and our attacker model. This will result in a major update in the next future. Finally, after fixing the vulnerability, one of the vendors asked us not to appear in our research.

¹⁴<https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>

Camera Fingerprinting Authentication Revisited

Dominik Maier¹, Henrik Erb², Patrick Mullan², and Vincent Haupt²

¹Technische Universität Berlin

²Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

Authentication schemes that include smartphones gain popularity. Instead of storing keys in app-private storage — cloneable by privileged malware — recent research proposes authentication with hardware fingerprints, arguing they will be harder for attackers to fake. Notably, the use of camera sensor fingerprints has been discussed recently. This paper revisits the eligibility of this camera sensor noise for authentication. The so-called *Photo Response Non-Uniformity (PRNU)* exploits use of production tolerances in the CMOS sensors, commonly used in smartphone cameras, to trace a photo to a specific phone and authenticate its user. We conducted the first large-scale study for PRNU on smartphones, with 56,630 images stemming from individual 3,809 devices across 1036 models. Based on the collected dataset, we reproduce proposed authentication schemes and uncover caveats not discussed in prior work on authentication. In addition, we give constraints an image used for authentication schemes needs to fit, to increase the reliability of the results. We are able to provide novel insights, implement attacks against the proposed schemes and discuss future improvements.

1 Introduction

With the emergence of smartphones, users carry around powerful multipurpose computing devices. As these devices are highly personal, identifying the smartphones typically makes it possible to identify the respective owner. For certain use cases, for example banking apps, smartphone apps replaced true second factors, such as hardware tokens. Some modern banking apps and the start-ups behind them value user experience and time to market over security [27] — and the majority of users seemingly prefers apps over purpose-built hardware devices they have to buy and may lose. In contrast to purpose-built hardware tokens without internet access, on phones privileged malware can misuse any secrets stored in apps. Haupt and Müller show that such app-based transaction schemes for banking — even those relying on two separate devices — can be attacked [29, 30]. Instead of storing

secrets that can always be copied off [28], apps can calculate unique fingerprints of phones on the fly. The idea of device fingerprinting is simple: production tolerances in sensors and other input hardware are assumed to be unique enough to be utilized as identifying key for each individual device. Fingerprinting of browsers and mobile phones alike, is already used extensively for marketing purposes [57]. In recent years, researchers and vendors also adapt fingerprinting schemes to identify and authenticate the user and the device. One well-suited sensor is the phone’s camera. On photos taken, the imaging sensor leaves imperceptible noise that can be used to identify the respective camera. Ba *et al.* [4] as well as Valsesia *et al.* [56] constructed authentication schemes based on camera fingerprinting. These schemes work on any current smartphone without additional hardware and therefore without impacting usability. Attackers, on the other hand, may also be able to learn the respective fingerprints and abuse them for malicious authentications. In this paper we take a close look at the schemes proposed. We revisit the idea of camera fingerprinting and put effort into providing an in-depth answer to the question, if and how camera fingerprinting enables secure smartphone authentication. An adversarial mindset and the results of a large-scale study unveil drawbacks in camera fingerprinting for authentication. Notably, we provide simple attacks against each attack detection step Ba *et al.* [4] present.

Contributions

In detail, we make the following contributions:

- Our evaluation provides an in-depth view on PRNU camera fingerprinting for authentication. As part of our research, we gathered images from 3,809 unique smartphone devices yielding 56,630 pictures, from 1036 models running both iOS and Android. To the best of our knowledge this is by far the largest dataset of images from smartphones recorded under a controlled environment, e.g. our self-implemented app, allowing for a realistic security assessment. With this, we were able

to reproduce and proof prior assumptions and published results.

- We introduce a forgery resistant camera authentication scheme with potential real-world security benefits, even on phones where keys can be securely stored.
- We present realistic attacks against proposed authentication schemes based on camera fingerprinting. We uncover flaws in defenses of the ABC protocol proposed by Ba *et al.* [4]. Most notably, we elaborate a replay attack and two fingerprint forgery attacks.

2 Background

First we provide background and related work around device and user fingerprinting as well as authentication. We discuss a wide variety of ways to fingerprint devices and users to then dive into details about camera fingerprinting.

2.1 Device Fingerprinting

For the web, fingerprinting users and their browsers in a privacy invading way is well researched and widely adopted for advertisements and tracking [1, 14, 46, 57]. This trend also emerges on phones, where an even larger variety of sensors can be leveraged. Mobile device fingerprinting through apps is of ongoing research interest. In the following, we present the state-of-the-art in device fingerprinting for mobile platforms.

Fingerprinting by Exploiting Properties of Sensors Prior research proves that most hardware sensors on phones can be used to fingerprint specific devices. Yue [60], Das *et al.* [13] and Bojinov *et al.* [6] have all used position sensors, accelerometer and the gyroscope, to fingerprint phones and their users. Hupperich *et al.* [31, 32] go one step further and use the available sensor data for an authentication scheme based on the device fingerprint. Das *et al.* fingerprint the devices using microphones [12]. Zhou *et al.* [62] and Das *et al.* [12] use speakers and microphones modules to uniquely identify a smartphone.

Fingerprinting by Exploiting Human Characteristics Fingerprinting the user directly, recognizing differences in individual movements and behavior, for authentication, is another theme. Nickel *et al.* recognize the users' walking patterns to authenticate them [45]. Frank *et al.* track specific touch behaviors for authentication [17]. Gong *et al.* propose a forgery resistant tracking method for user touches [25]. Bo *et al.* [5] combine touch and sensor fingerprinting to authenticate specific users. Instead of recognizing input, Kurtz *et al.* [35] and Wu *et al.* [59] leverage user settings and files to track iOS and Android devices with high accuracy. Further proving users' actions can be used for unique identification,

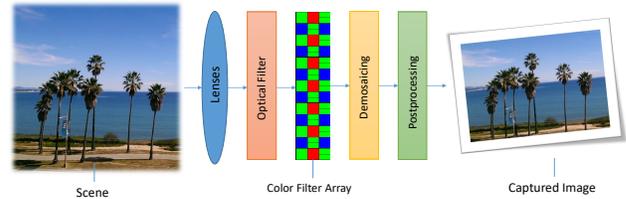


Figure 1: Digital Camera Image Pipeline

Stöber *et al.* [54] identify smartphones by monitoring the network traffic from Facebook, Skype, WhatsApp and Dropbox for 15 minutes. Reaves *et al.* [51] propose the use of phone calls to authenticate the user remotely.

2.2 Smartphone Camera Fingerprinting

With camera fingerprinting, this paper focuses on anomalies in the CMOS sensor of smartphones. Before light reaches the CMOS sensor of a camera, it passes through multiple elements. All of these, as depicted in Figure 1, introduce imperfections specific to a device or make. Light travels through lenses, an anti-aliasing filter and the color filter array. Typical parts of the image pipeline are discussed by Ramanath *et al.* [50]. Like all elements in the pipeline, and other sensors in a phone, the CMOS sensor of every camera is subject to production tolerances. The phone camera's megapixels translate to the number of colored dots it captures. Every pixel in the final photo then consists of intensities of the three colors red, green and blue. These intensities are the amount of light reported by the CMOS sensor at this position. Some CMOS sensor elements will systematically interpret the light impact more or less intense by a slight margin, that might exceed the granularity of discretized pixel values. Ultimately, this leads to a peculiar pattern that can be found with varying strength throughout all images of a given imaging device. This pattern is called *Photo Response Non-Uniformity* (PRNU). Even though, this pattern is weak, usually imperceptible to the human eye, methods have been developed to extract it reliably from images [22, 38]. Further, these works showed as a remarkable property of PRNU that it is highly unique across different devices. This holds even for different devices of the same brand and model. We focus on PRNU in this paper even though CMOS Image Sensor Fixed Pattern Noise is also unique, as discussed by Kim and Lee [34], as it works with brightly lit, captured photos. Lukás *et al.* [38] conclude that other sources of noise, like fixed-pattern-noise and shot noise, are not as well-suited.

For the mathematical explanation of the PRNU, refer to Appendix A. The PRNU was shown to be reasonable stable over the lifetime of a camera [19] and provides a solid way to associate an image to its source device. For the sake of completeness it shall be mentioned that, for our study in Section 3, further post-processing of the estimated fingerprint is done

by our C++ code. Specifically, the code removes non-uniform artifacts, as other implementations do as well. As these optimizations are not fundamental to the concept of PRNU, we do not elaborate on them here, but refer the interested reader to literature like [21].

To verify if a new image was taken with a specific camera, the residual of the new image is calculated as shown in Equation 2. Then this residual is correlated against a reference fingerprint extracted of the camera in question. With the correlation we obtain a value expressing the similarity between the reference fingerprint and the new residual under test. As correlation metric, the normal pearson correlation could be used. However, the *Peak Correlation Energy* ρ has shown to be a more suitable option for this application [21]:

$$\rho_{[I, \hat{K}]} = \text{PCE}(W_i, I\hat{K}) . \quad (4)$$

Based on this mathematical foundation, a considerable amount of research went into improvements to increase the quality of the extracted noise pattern [9, 23]. Furthermore, several attacks and counterattacks were discussed. Entrieri and Kirchner [15], Karaküçük *et al.* [33], as well as Li *et al.* [37] show how erasing and spoofing of camera fingerprints is possible and can be mitigated.

2.3 Camera Fingerprinting Authentication

We will briefly discuss two authentication protocols based on PRNU. The idea for both schemes is to authenticate by the unique fingerprint of a smartphone camera.

The elements commonly present in camera fingerprinting authentication schemes are as follows:

Elements

The Verifier

The verifier poses an initial challenge to the smartphone and verifies the final decision.

The Terminal

The terminal can be any device displaying the challenge (QR-Code) to the smartphone. It could either be a POS terminal, a computer screen or some other display and will be fed by the verifier.

The Smartphone

The device to be authenticated. The smartphone takes a photo of the terminal, adding its unique PRNU fingerprint to the image in this process. It will forward the final photo(s) to the verifier.

ABC According to the ABC scheme by Ba *et al.* [4], the verifier and terminal can be implemented on one device. The verifier needs access to a database of all registered phones. The basic interaction between all elements is depicted in Figure 2 and is as follows:

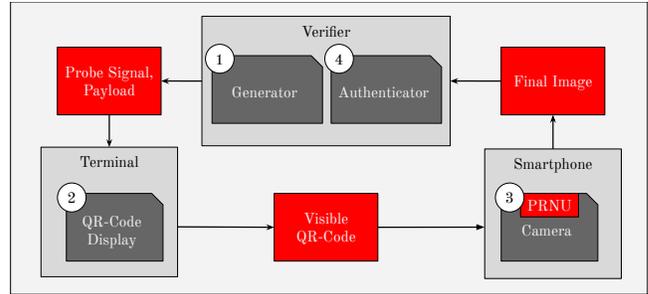


Figure 2: Camera Authentication Building Blocks

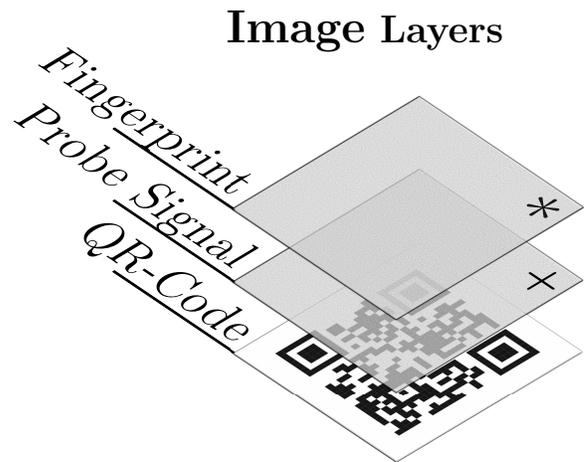


Figure 3: Layers of an image used for ABC authentication [4]

- ① The *Generator* creates an image with a QR-Code. The QR-Code contains transaction details and a timestamp. Then the probe signal is added to the image and sent to the terminal.
- ② The image is presented to the user on a display on the *Terminal*.
- ③ Agreeing with the content, the user captures an image of the screen containing the QR-Code and the probe signal. As a side effect of the image capturing, the PRNU of the camera sensor is added to the photo. The final image is then sent to the authenticator.
- ④ The *Authenticator* extracts the fingerprint of the final image and compares this to the reference fingerprint of the user stored at the server. If the PCE value is above a certain threshold, the transaction is authenticated.

Based on the apps created for our study, discussed in Section 3, we were able to implement the ABC protocol discussed hereafter. It is based on a registration phase and an authentication phase. For the registration phase, the user uploads one image taken through the smartphone. Ba *et al.* [4] claim that

one image for extracting the reference fingerprint is sufficient. During the authentication phase, the verifier challenges the user to take two images of a screen with two different QR-Codes. Both QR-Codes contain a time stamp and an excerpt of the ongoing transaction. The user then verifies the information about the transaction shown on the smartphone’s display, see Figure 2. An additional *probe signal*, designed to survive photographing but not fingerprint removal, gets added to the QR-Code displayed on the terminals’ screen. It is specified as *additive* white gaussian noise with a standard deviation of 5. The fingerprint in contrast, is *multiplicative*. A schematic illustration of this pipeline is depicted in Figure 3. The phone then sends the two photos to the verifier. The verifier determines the PCE value between the two images and each image with the reference image. If both values are above a certain threshold and equally high the verifier authenticates the user’s request.

RAW vs JPEG Another camera fingerprint-based authentication protocol was proposed by Valsesia *et al.* [56]. They also use the PRNU as physical unclonable property. For their authentication scheme, they assume that only the user or authentication app will have access to the RAW image data of the smartphone. All publicly available images are compressed and, therefore, do not contain the high-frequency components of the fingerprint of a RAW photo. Valsesia *et al.* [56] introduce a way to compress the fingerprint using random projections. This reduces their size and brings a big advantage for transferring and storing the PRNU of multiple cameras. Also, the raw fingerprints do not need to be sent over the network completely. Secret side information for the random projections never leaves the side of the smartphone. The server does not store the compressed fingerprint itself, but uses a fuzzy extractor scheme to create a uniformly random bit string. This prevents an adversary, who gains access to the server, to obtain the stored fingerprints of all users easily. Quiring *et al.* [48, 49] propose the use of Fragile Camera Fingerprints, proofing the fingerprint cannot be recovered from JPEG images, but RAW images alone. To attack these schemes, access to the phone is required, as RAW images rarely get posted to social media.

3 Large-Scale Data Acquisition

To study the real world applicability of the proposed schemes, we implemented a setup for authentication and evaluated it on thousands of images. The dataset was collected from scratch, using apps that resemble implementations of the proposed authentication solutions. We describe here how we designed our apps but first discuss advantages of our new extensive dataset in general, and compare it to other already existing datasets.

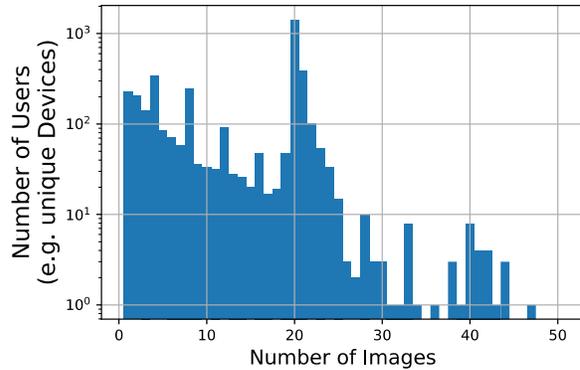


Figure 4: Number of images for individual devices

3.1 Methodology for Data Acquisition

According to Zhang and Zhang [61], around 20 images as training set are a good balance between image count and strength of the PRNU. Our initial tests on a small number of phones showed that additional images still increase the PCE of generated patterns against new images. The PCE for 20 images was usually above the threshold of 60, proposed by Goljan *et al.* [21]. We decided to let participants in our study capture and upload around 20 images to learn their individual PRNU pattern. User were free to quit the study before completion and could upload more images, if desired. This explains why the total number of images is not a multiple of 20. To minimize the workload for participants further, one click on the trigger captured batches of 5 photos in a few seconds of time. Even a small delay between subsequent images within a batch drastically increases the changes of two back-to-back images being unaligned [58]. The apps save JPEG files in full resolution and default quality offered by the underlying platform. In Figure 4 the distribution over number of images per unique device is shown. Even though our study thoroughly informed the user about privacy implications and asked to never upload photos of persons, we will not publicly release the dataset at this point, as proper vetting of 56,630 images is infeasible. For reproducibility, access to the dataset can be made available upon request.

Our large dataset is tailored for investigating authentication schemes on mobile phones, and hence surpasses other image forensic database for this purpose. For instance the large dataset of Goljan *et al.*’s study [21] consists of samples from photo platforms on the internet, however our dataset was only collected through our auth-app, on which we have full control. This tight grip means no (unknowingly) cropped, edited or otherwise altered images impact our training set negatively. The Dresden Database [18] is another image database, popular for image forensic works, containing images from 73 unique devices sampled from 27 models from 4 manufacturers. Both datasets were collected prior to 2010, meaning on

images taken with classic cameras, e.g., not contemporary smartphones in the context of mobile authentication. The VISION database [52] was tailored for mobile devices, however is rather small in comparison to our dataset, with only 35 devices. RAISE [11] is another database often used in works focusing on image forensics. It was designed to offer high-quality raw images from 4 professional DSLR cameras, hence also not suitable for our purposes of vetting PRNU based smartphone authentication schemes.

The authentication platform implemented for our study and the backend consists of three main components, apps for Android and iOS, the Database Server, and a worker server to run PRNU comparisons and evaluations asynchronously.

The apps for iOS and Android can be compiled in different modes: *large-scale study* or *authentication*. In large-scale study mode, both apps for iOS and Android inform the user about the study, goals and the types of photos they are supposed to upload. It provides live information on how their camera compares to that of other phones, including median, mean, maximal and minimal PCE of the user phone’s PRNU against other images. This mode was used to gather data about the fingerprinting effectivity and to collect the dataset.

On the workers, the C++ core to extract the PRNU and compare it to images, *MagicFern*, is a speedy reimplementa-tion of the popular Matlab framework by Goljan *et al.* [21] available on their website. It is quick, due to the low level implementa-tion and multi-threading. We benchmarked our tool against the Matlab implementation to ensure we achieve a same level of accuracy. All parts of the platform will be open sourced upon publication.

3.2 Participants

All users participated voluntarily in our large-scale study. The app was available to the public and announced on social media and news outlets. First, it informs the participants about all processes transparently. Then, it instructed them to upload different scenes and never to upload personal or identifying images. The only information gain associated with the image and PRNU is, if a phone participated in our study. Although we advised users to not photograph private entities, like other people’s faces, we can not guarantee all users complied due to the large amount of uploaded photos. We will therefore not release the images publicly.

The large-scale study had a total of 3,809 participants from both major mobile platforms over the course of one month. In a platform breakdown, 25% of patterns were learned on iOS, 75% on Android. In total, 1036 different models from 137 manufacturers took part in the study. The top three brands were *Samsung*, followed by *Apple* and *Google*. In total, we collected 56,630 images.

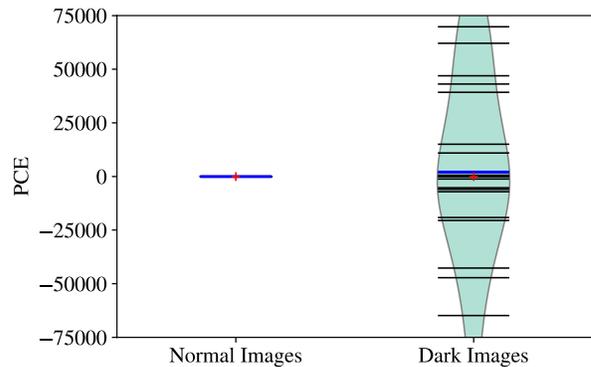


Figure 5: Distribution of estimated PCE values for two datasets. One dataset has only dark images the other one arbitrary sampled images. Further, all images were picked to *not* match a reference fingerprint. Normal images exhibit a low PCE score, as desired. Dark images in turn behave arbitrary, and often even have a large positive value.

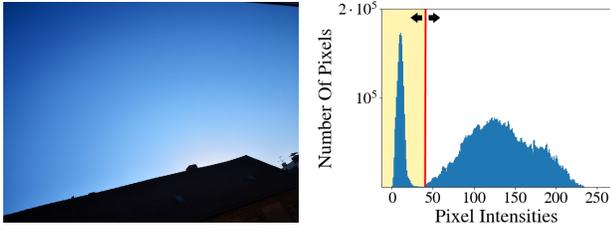
4 Evaluation

Based on the study, discussed in Section 3, we evaluate and rate the results we obtained from these 56,630 unique images of 3,809 smartphones. We look into the quality of the fingerprints on smartphones and determine constraints to obtain a fingerprint of the highest possible quality. We examine the behavior of the PCE as correlation measure for images and fingerprints under varying characteristics. Images recorded by the same camera should match, hence need to have a high PCE value. Images from different cameras should not match. Their PCE value should be close to zero. Since smartphone cameras produce images of different resolutions calculating the PCE value can be tricky. A common way of overcoming this problem is by cropping out a fixed size patch from all images [21, 23, 24, 36]. We picked a squared patch of size 1024 pixels always centered.

4.1 Effect of Illumination on PRNU

We examined the bad influence of known issues for noise extraction and noise correlation to evaluate the importance of the different constraints. In general, extreme PCE values appear with photos that are too saturated. A setting that is known to be challenging, or even preferable to avoid, if possible, for fingerprint estimation [9, 23].

Influence Of Dark Areas We reproduced previous re-search [22] showing a clear correlation between image il-lumination and the PCE value. Our evaluation script inspects all images from each user and compared them to all other im-ages of this user/phone. This should always yield high PCE



(a) Photo with large homogeneous areas and a small local standard deviation of 0.006. (b) Corresponding histogram to the homogenous photo.

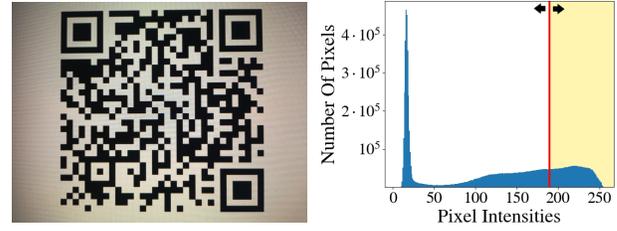
Figure 6: Typical scenery image and histogram of its pixel intensities.

values, because they were captured with the same camera, but this oftentimes fails for very dark images. In Figure 5 we plotted the distribution of 21 comparisons of a camera fingerprint from an arbitrary picked phone against random images of different phones by determining the PCE value. The values of normal images are to be expected. They are close to zero with small variance. In strong contrast, the other dataset shows the PCE values of the comparison of only either dark or completely black images from different cameras. The absence of any noise information sometimes randomly triggers very high positive or negative correlations. They generate almost arbitrary PCE values.

To improve the results, we then only took images which met certain constraints, namely an intensity threshold. In Figure 6b we show the histogram of Figure 6a. The varying threshold is represented as the red line. From every image the percentage of the pixels with an intensity below the threshold was determined. This percentage of pixels is illustrated as the yellow area. After this, we varied the constraints an image needed to meet for being taken for fingerprint extraction.

The question arises if there is some sweet spot for the intensity of the illumination of an image or if brighter is always better. In theory, above a certain brightness the pixels of an image are saturated and the fingerprint attenuates. Therefore in the next step, we examined the dependency of the PCE value under the constraint of brightness.

Influence Of Bright Areas The images used for determining the PCE values again needed to fit two constraints. A varying amount of pixel intensities needed to be above a varying threshold. This time we chose a varying threshold from 155 to 255. The threshold is illustrated as the red line in Figure 6b. The yellow area represents the pixel intensities for determining the percentage. The histogram of Figure 6b belongs to the image of Figure 6a. One image per user was compared with all other images of the user, if they fitted the given constraints. From this, the median of all PCE values was determined. Every dot in Figure 8 represents one of the medians. Again we used 100 users and determined the mean over all PCE values per varying threshold. The mean values result in a quadratic



(a) Photo of a QR-Code. The standard deviation of 0.174 is very high. (b) Corresponding histogram of QR-Code image.

Figure 7: Typical Image of QR-Code to forward transmission details in an online banking scenario as suggested by [4]. To the left the corresponding histogram of pixel intensities.

curve with a maximum at 173. Therefore we can conclude that one obtains the best results for images that should match, if the threshold for the brightness of the pixels is at 173.

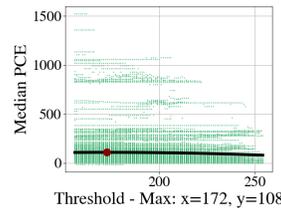


Figure 8: Median PCE for a set of images, where a percentage of pixels is above the depicted threshold (155 to 255)

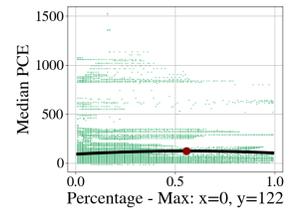


Figure 9: Median PCE for a set of images, where the depicted percentage of pixels is above a certain threshold (155 to 255)

Figure 9 shows the same data as Figure 8. The difference lies in the fact that we look at the varying percentage of pixels with an intensity above a certain threshold. The mean over all PCE values per percentage also result in a quadratic curve. The maximum is located at 56%. We can observe that it is not useful to have images as bright as possible but rather relatively high.

Perfect Ratio Between Percentage And Threshold As a further step we optimize for the perfect ratio between the chosen percentage and threshold. If we vary both variables at the same time we obtain a surface. We varied the percentage of the pixels that need to be above a certain threshold from 0% to 100%. In addition we varied the threshold the pixel intensities needed to be above, from 155 to 255. We then used a box linear filter with the kernel size of 9 to smooth the PCE peaks to make a more general statement. The resulting surface is presented in Figure 10. The global maximum lays at a percentage of 100 and at a threshold of 197. The plot confirms that, with increasing percentage, the PCE value increases as well. However, as depicted in Figure 8, the PCE stops increasing and even decreases for very high values. In

Figure 10 we see, that it falls off very fast after reaching the maximum. Too bright images produce very bad results for fingerprint correlation.

Concluding from this, the more areas of an image contain a high pixel intensity, the better the fingerprint extraction works. To obtain a fingerprint with the highest possible quality, images with 100 percent of the pixel intensities above 197 are the best fit. This means, a possible authentication scheme can optimize screen brightness on the one hand, and reject bad images on the other.

4.2 Scenery as Constraint

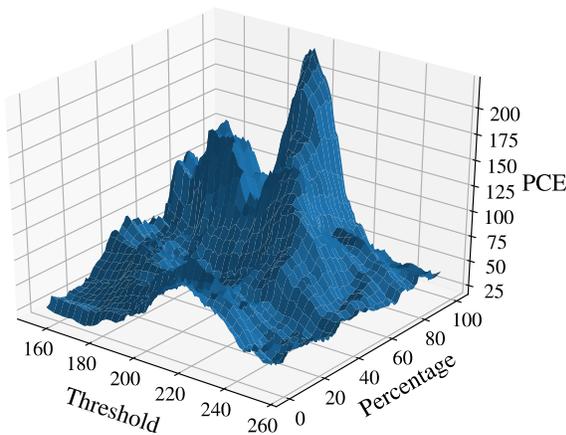


Figure 10: Surface plot describing the behavior of the PCE of images, which meet the constraints of the respective threshold and percentage. A sweet spot is found if most pixels are at an intensity of ≈ 200 .

The photographed scenery impacts the quality of the extracted PRNU [9, 23], as well. The ultimate goal of the high-pass filtering, as described in Equation 2, is to suppress the image content. Nevertheless some content, even if weak, always remains, and leaks into the noise residual. This affects the performance of authentication schemes.

From every image of our study, we determined the standard deviation of every pixel intensity to its eight neighbors. We then calculated the mean overall standard deviations and used this value for further comparisons. As in the calculations before, we calculated the PCE value from one image of a user with all other images of the user. Then, we constrained the images used for PRNU extraction to a mean standard deviation above a certain threshold. In Figure 11, all dots of a particular color belong to one user. The plot shows 60 different users. With each dot the constraint of the standard deviation was increased to the point that one image dropped out of the set. In most of the sets of the users, this resulted in an increase of the mean over all PCE values of the images.

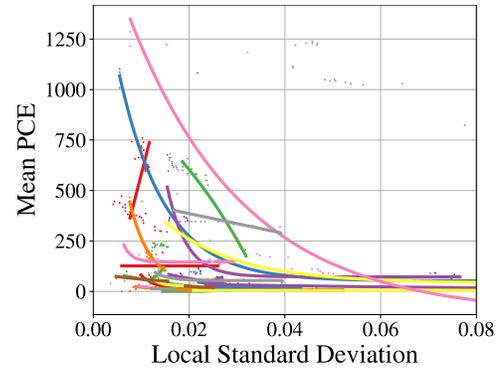


Figure 11: Calculated PCE value as a function of local standard deviation of pixel intensities in an image. The general trend of the downwards sloping curves show that variance impacts the magnitude of PCE.

Furthermore, we fitted a curve into the point clouds. The behavior of the PCE values over a decreasing standard deviation seems to be exponential. We can therefore conclude that the local standard deviation of an image is crucial for PRNU extraction. Homogeneous areas in an image increase the quality of the extracted fingerprint. This results in a higher PCE value for images that should match. Therefore, filtering images under the constraint of the local standard deviation results in a strong increase of the true positive rate.

The local standard deviation of the images of Figure 6a and Figure 7a differ a lot. The QR-Code with a lot of edges and a strong Moiré pattern has a local standard deviation of 0.174. The image of the sky has a local standard deviation of 0.006, which is extremely low, in fact about 30 times less. Therefore, it seems that images containing a QR-Code are not optimal for fingerprint extraction.

5 Discussion

In this section we discuss the impact of our results for camera fingerprinting as authentication scheme. We look into flaws in the authentication protocols based on our findings. We present several possible attacks and defenses.

5.1 Images *DO* Need Constraints

Ba *et al.* [4] do not discuss possible constraints for photos taken for their ABC scheme. As we show in Section 4, the photo content has a strong impact on the PCE quality. Figure 11, for example, shows that the PRNU extraction is negatively affected by high frequency scenery in the image.

Also, concluding from Section 4.1 we can say that the brightness plays an important role for obtaining a good fingerprint resulting in a low false negative and low false positive rate. Images with 100% of the pixel intensities above 197

yield the best results. Since cameras adapt the brightness automatically, taking such a photo might only be possible in extreme lighting conditions. The QR codes used in [4] are default black and white prints. Therefore, depending on the exact exposure and recording, the color settings in such QR codes are prone to produce pictures that are unsuitable for proper PRNU analysis. To improve the true positive rate of an authentication protocol even further, one could add a function for determining the local standard deviation of the challenge and reference images. From Section 4.2 we know that this improves the quality of the extracted fingerprint as well. With the mentioned constraints, we can essentially increase the reliability of an authentication protocol.

Also, we can conclude that rejecting images for fingerprint extraction with a high local standard deviation can decrease the false positive rate and false negative rate. Images with a low local standard deviation in many areas of the image are suited very well for fingerprint extraction. The sample images in Figure 6a and 7a depict histograms of comparable shapes, as seen in the neighboring Figures 6b and 7b. However, the images differ significantly with respect to local variance. The sky image has large flat areas, the QR image has many transitions from black to white. The sky image has a very low standard deviation of 0.006, but the QR image has a standard deviation of 0.174. Looking at the plot in Figure 11 we see, that with increasing standard deviation, the quality of the PRNU decreases exponentially.

5.2 QR-Codes for Fingerprint Extraction

The ABC paper does not specify the size of the QR-Code relative to the whole image [4].

We find that images containing QR-Codes have a high local standard deviation. It is almost impossible to extract a correct PRNU from an image area containing a large QR-Code as scenery. The scenery of an image containing a QR-Code has high frequencies: The black and white QR-Codes consist of lot of edges, which leads to very high frequency components in the image. Further, the QR images were taken by capturing the QR-code from a LCD-screen of a monitor. This induced a Moiré pattern. Since the PRNU also is a high frequency signal, it is therefore not possible to differentiate between the scenery and the fingerprint. This leads to artifacts and a deterioration of the PCE value. To somewhat mitigate this problem, our research shows that using grey and white QR-Codes for fingerprint extraction improves the method. For our proof of concept we used a Huawei P20 Pro. The resulting PCE values for images taken by the same camera were five times higher when using grey QR-Codes compared to black QR-Codes. Therefore, we can conclude that grey and white QR codes, while, still being accepted by QR code readers, result in PCE values better suited for authentication, probably as even the grey area still produces enough light for the sensor to add a strong PRNU pattern. The contrast and therefore the

edges are not that strong, compared to black and white, and the dark areas still contain light, thus resulting in an increased PRNU.

5.3 Fingerprints Require Multiple Reference Images

The user, in the scheme proposed by Ba et al. [4], only requires the user to record a single image with a smartphone. The image then is sent to the server, which extracts the fingerprint from this single image and stores it for later authentication processes with challenge images.

Assuming that images of smartphone cameras are well fitted for PRNU extraction, we have doubts on the assumption that one image is sufficient for creating a high quality reference fingerprint. To our knowledge, this opposes the prevailing opinion in literature. One of the key assumptions in extracting PRNU is that other noises, like shot noise or random noise, can be averaged out over a corpus of N images, as Equation 3 states. In original works [38] up to 50 images were suggested. This has been lowered to 20 by some authors, eg. Zhang et Zhang [61]. With only one image at hand, other sources of noise will essentially persist throughout fingerprint extraction. Our experiments discussed in Section 5.1 suggest that scene content plays a big factor, may it be, because the overall images are too dark, or because they contain too much high frequent content, in textured areas. To leverage this, some authors even suggest to obtain the fingerprint exclusively from images without texture [21]. In our experiments we relaxed this condition and considered also images with texture for extraction, and still got reasonable good results. Nevertheless, in an authentication scheme, may it be for online banking, we argue that no risk should be taken and a solid number of high quality images with little standard deviation should be used for creating the initial fingerprint.

6 Practical Attacks

First, we will introduce the assumed threat model, then we will present three attacks on the ABC scheme, as we implemented it based on [4].

6.1 Threat Model

The attacker assumed by Ba *et al.* [4] is, in our understanding, rather weak. The threats do not factor in malware, which could take photos on the phone and learn fingerprints with ease, even though malware on smartphones is rather common [39,47]. The adversary may gain access to public photos of the victim, is able to sniff the communication channel between victim and verifier, and can fake a display the victim takes images of (phishing). It includes replay attacks, fingerprint forgery attacks (adding a learned fingerprint to photos not taken with the original smartphone) as well as *Man in the*

Middle (MitM) scenarios. The goal of the adversary lies in convincing the victim to authorize a malicious request, or to trick the verifier into authenticating the adversary's request.

The scheme by Valsesia *et al.* [56] relies on an attacker that will not be able to capture RAW images. As this even rules out unprivileged malware with camera access, in our further discussion we focus on the scheme of Ba *et al.* [4].

We assume the following threats.

6.2 Fingerprint Forgery

A well-known approach to tackle fingerprint forgery is to use the triangle test [37]. Ba *et al.* [4] rightfully desist from using this technique on the base, that it comes with high computational effort. A database with all public images for each user would have to be curated and updated constantly. They present a novel technique based on the recording of two images. During the authentication process, the user uploads not one but two images of two different QR-Codes. If an adversary applies the victim's fingerprint on both photos, they still carry their real fingerprint. Hence, their extracted fingerprints have a higher correlation than the comparison with the reference fingerprint stored on the server. By this, Ba *et al.* [4] detect an additional fingerprint on an image. Even if the attacker were not able to erase fingerprints completely, for which multiple ways have been proposed, for example by Bonettini *et al.* [7], attacks are still possible. Using two different smartphones, a single dual camera phone or even just rotating the phone by 90 degrees breaks this defense. For our proof of concept attack, we cropped two, mostly disjoint (they still need to contain the QR-Code), regions from the same image sensor. Different parts of the same sensor have a different PRNU as well.

6.3 Image Reuse

Despite the addition of QR-Codes, we are still able to reuse any images from the victim's phone, thanks to the noisiness of QR-Codes. Due to this high noise, large black areas and the many high-frequency edges of the scenery, as discussed in Section 5, our tests show QR-Codes are almost ignored during PCE calculation. We can use this fact to our favor. After taking a photo of the QR-Code with a second smartphone, including the probe signal (see Section 6.4), we crop the QR-Code from the image and paste it into the victim's image. The QR-Code has to be as small as possible but still large enough to be accepted by the server and containing enough of the probe signal. Because of the small proportion of the QR-Code compared to the whole image, the area containing the fingerprint of the victim is still very big, with the QR-Code having almost no impact. The correlation of the manipulated challenge image and the one stored on the server is high. We were able to create false positive results against our proof of concept implementation this way. It is hard to defend against

this attack without adding some sort of image recognition or different forgery detection: if the photographed QR-Code needs to be too big, its noise can result in a high false negative rate, as the fingerprint may be blocked. Addressing this vulnerability, we present a countermeasure in Section 7.3.

6.4 Probe Signal Preserving Fingerprint Removal

In the Section 6.2, we showed that two parts of the image are already enough to forge a foreign fingerprint without failing the authentication process presented by Ba *et al.* [4]. A more elaborate attack first erases the fingerprint of the adversary's camera to then add the fingerprint of the victim. To prevent this, Ba *et al.* [4] introduce an additional probe signal. They specify it as white gaussian noise. This probe signal is applied to the QR-Code displayed by the terminal.

With a black-and-white QR-Code image as base, we assume the signal to be additive. In Figure 3 the resulting layers of noise are depicted. The QR-Code represents the scenery, perceived by the user, containing a lot of high frequency components itself. The probe signal symbolizes the additionally added white gaussian noise. The authors claim that the noise with a standard deviation of 5 is of the same variance as a fingerprint. The top layer represents the fingerprint of the camera sensor, the PRNU. It is added during the recording phase of the image by the varying sensitivity of the camera sensor. Our large-scale test indicates that the standard deviation of the PRNU is a lot lower than 5. A deviation of 5 would result in images with visual noise.

The idea of the probe signal is, that it will be erased if the adversary tries to erase her own fingerprint from the image by using a low-pass filter. If the probe signal is not contained in the challenge image, which is compared to the reference fingerprint and to which the probe signal was added as well, the correlation decreases. By this, Ba *et al.* [4] claim to detect any erasing of a fingerprint from an image.

Our analysis shows that the challenge images can be analyzed with a high-pass filter, to find all noises — both the noise due to the probe signal, as well as the PRNU noise. We can further estimate the fingerprint of the device, for example from other pictures taken on it. With that, we are able to dissect the probe signal from the inherent fingerprint, and clean the attack. This works even if we assume that both signals are of similar strengths. However, on our setup, the noise of the PRNU is much weaker. So simply filtering strong perturbations, with respect to the fingerprinting, suffices. The underlying problem here is, that the probe signal can be considered as a watermark, which is by design detectable, even if the detection should be done on the server.

7 Beyond Camera Fingerprinting Authentication

Besides improving the PRNU extraction in general to improve the significance of the results as proposed in 4 one can undertake further steps. In this section, we propose a transaction security scheme that takes the current state-of-the-art into consideration and improve upon the drawbacks.

7.1 General Camera Fingerprint Improvements

Any fingerprint can be learned by attackers, then be replayed or dropped directly in authenticating code. This is still possible — although harder — after applying probe signals and checking multiple uploaded images, as proposed by Ba *et al.* [4]. Still, attackers crawling images or malware on the phone, can learn the fingerprint and forge correct images to authenticate. Lukás *et al.* [38] note that it is “unlikely that there exists a numerical identification characteristic computed from digital images that could not be compromised by a sufficiently sophisticated opponent”. Malware on the defending phone can simply capture 20 images to gain the statistical pattern and then learn the fingerprint. The malware then can create a photo that passes any other requirements, for example including the correct QR-Code for authentication. It can also remove any fingerprints and apply the learned fingerprint on top. However, defenses against fingerprint forgery (and counterattacks) are an ongoing research topic for image forensics. As it is common in the field of security, we are subjected to an arms race between attacker and defender. Trivial anti-forensics for camera fingerprinting have been broken early, for example by the so called *triangle test* proposed in 2010 by Goljan *et al.* [20], which can be attacked yet again [40]. The arms race in camera fingerprint forgery detection is ongoing with new defensive methods being proposed more than ten years after the paper by Goljan *et al.* [55]. As an alternative to ABC, Ba *et al.* [3] recently proposed CIM, a scheme that combines additional noise in multiple burst photos, as well as accelerometer fingerprinting for a more resistant fingerprint. Additional sensor inputs during photography could even increase the attacker’s effort in cloning in the future. For applications that need very high security standards, an authentication scheme could go beyond checking the fingerprint for correctness and also detect possible forgeries on the image itself [10].

7.2 Trusted Secure Camera

Our study shows that the camera fingerprint poses little real world use and leaves room for attackers. The current mitigation approaches proposed by Ba *et al.* [4] can be circumvented (see Section 5) and further defenses can only lead to

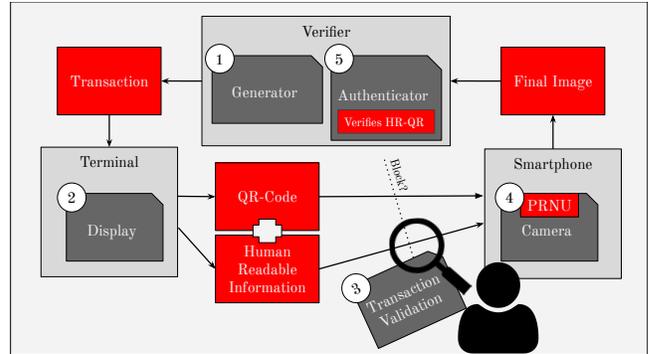


Figure 12: Illustrated scheme of an updated authentication protocol. The user can always cancel the ongoing transaction. The terminal and smartphone can both be adversary-controlled.

an arms race that the defender will lose. However, device authentication through images could substantially be increased if the camera output could not be forged. As a matter of fact, cameras that introduce unforgeable cryptographic assets in a captured image have been proposed before, for example *PhotoProof* by Naveh and Tromer [44]. If smartphones were to implement such a secure camera module, secure image based transaction schemes can be implemented. These cameras, however, are state-of-the-art research objects, expensive, and will hardly ever make it into smartphone hardware.

On the other hand, trusted execution environments are steadily gaining ground and modules exist in most mobile devices today. Direct access to external hardware is also already feasible, e.g. in ARM TrustZone [2]. Ultimately, by hooking up trusted software to the camera directly, a system for unforgeable camera fingerprints can be built as long as an attacker cannot gain access to the TEE. Images (or videos) taken by this *Trusted Secure Camera* could embed watermarks or signatures like *PhotoProof* [44] inside the image. Another approach is to attach a signature in the metadata before the image leaves the trusted and attested code. An ideal solution only allows write access to the memory region the camera data is written to by the hardware and secure camera stack in trusted code. This *Trusted Secure Camera* would render normal camera fingerprinting redundant and help secure transactions. Since our evaluation shows that almost all images are traceable, drawbacks in privacy are minimal and the feature could also be disabled by users if needed.

7.3 Securing Transactions

In this section we propose a forgery-resistant scheme for secure transactions. Whereas for authentication purposes the methods are flawed, it poses a real benefit for transaction signing. For this, we have extended our camera fingerprinting app built for the large scale study with a banking-like scenario.

In our proof of concept application, our signing app uses optical character recognition (OCR) with a QR-Code as redundancy to ensure that the user really saw the correct transaction contents.

The computer can display the TAN information directly instead of relying on a phone. If malware fakes the on-screen info, the user will immediately see it and is able to react. As depicted in Figure 12, if the content on the display does not match what the user wants to sign, a user can easily identify the faults. The user will stop the transaction automatically if the displayed data is wrong. The transaction does not get signed. The signature is linked to a user's intent. This results in cancellation due to not using the second factor at all. We can achieve this by linking the data displayed to the user in a human and machine readable format. The camera fingerprint then makes sure the user saw the photographed content at least once.

A similar solution to this kind of simple human readable QR (*HR-QR*), has been proposed by Millican and Stanjano [42]. Their so-called *SAVVicode* combines a QR-Code with a more machine-friendly text above, making it even easier to process without errors. As the codes, just as QR codes, are very noisy, they should be *grey* and *white*. The key components are as follows:

1. **HR-QR:** The user needs to know what she scans. The only way to do this is to show the contents in clear text. This human readable QR-Code is then checked by attested code.
2. **Authenticity is checked by extracting the PRNU:** The same attested code needs to check if the correct camera took the photo at hand, using the camera fingerprint. This will lower the risk that attackers directly insert images in the black box and run the authentication code without the user's knowledge. At the same time, an air gap is enforced, making the phone a true second factor.

The two key components are

- **Matrix code is human-readable:** To counter MitM schemes, the user needs to know what she scans. The best way to do this is to display the contents in clear text. We add OCR to a usual transaction scheme [41]. Since OCR software is not fail-proof enough, the content is sent along in a separate QR-code or other barcode or matrix code, like [8]. Trusted code, either in the TEE or on the server then checks if both parts match.
- **Authenticity is checked by extracting PRNU or Fingerprint:** The same, attested, code needs to check if the correct camera took the photo at hand, using the PRNU. This will lower the risk for attackers to directly insert images in the black-box and running the authentication code without the user's knowledge. At the same time, an air gap is enforced.

The camera has the merit, especially in the case of an *HR-QR*, that it contains additional information that can be linked to a user intent. The user will only scan the screen if she wants to sign the authentication. For this study, we implemented an *HR-QR* scheme. Details are discussed in Appendix B.

Everts *et al.* [16] note on their implementation of smartphone authentication scheme, that a secure element won't prevent malware on the phone to use the credentials. We argue that, using the camera fingerprint as input, this statement no longer holds true. Instead of feeding data to the secure element, the attacker now has to feed an image that satisfies all imposed restrictions. For the future we plan to replace PRNU with a secure camera, as discussed in 7.2. That means, as long as the trusted execution environment can be considered safe, *HR-QR* is then a secure transaction system, even with compromised smartphones and displays.

8 Conclusion

The attacks on current authentication schemes, presented in this paper, show the shortcomings of smartphone camera fingerprinting. We show attacks against defenses discussed by a major authentication protocol leveraging camera fingerprinting, allowing us to impersonate the user, even with a single photo. In our comprehensive large-scale study, collecting 56,630 images, we can substantiate that all current phone cameras work well as PUF, essentially pinning photos to unique devices. After thorough vetting, we can conclude that, in direct comparison with other authentication schemes, the added complexity of camera-fingerprinting based authentication does not add substantial security benefit over secrets stored in the phone's memory. While it does add benefits over weak schemes, such as SMS based authentication [43], possible alternatives exist: pushing an one-time password (OTP) to the phone at the time of authentication through a secured internet connection, through bluetooth or even via sound. In all threat models discussed in Section 6.1, normal app authentication schemes are either good enough, i.e., if there is no privileged access to local contents on the phone, secrets can be stored there — or the camera fingerprinting authentication schemes also fail.

Just as it is the case with fingerprinting in general, it might pose an additional hurdle for attackers, however, the knowledge how to calculate camera fingerprints is readily available. Apart from the special case where the defendant never discloses RAW images and uses them for authentication only, as proposed by Valsesia *et al.* [56], the security of camera fingerprinting is further damped by publicly available images. We do see a benefit for transactions as the photographed screen can replace a secure display. For the future, signing photos in the trusted execution environment of a phone could replace PRNU use-cases completely and provide security benefits.

We hope to enable fruitful future research based on our findings as well as the code published as part of this research.

Acknowledgments

The authors would like to thank Federico Maggi, Stefano Zanero, Tilo Müller, and Christian Riess for feedback and support.

Availability

The PRNU implementation in C++/OpenCV will be open sourced upon publication. Additional source code needed to reproduce results of this study, including our proof of implementation for authentication, as well as evaluation code, will be made available.

References

- [1] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz, “The web never forgets: Persistent tracking mechanisms in the wild,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 674–689. [Online]. Available: <https://doi.org/10.1145/2660267.2660347>
- [2] T. Alves and D. Felton, “Trustzone: Integrated hardware and software security,” *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.
- [3] Z. Ba, Z. Qin, X. Fu, and K. Ren, “Cim: Camera in motion for smartphone authentication,” *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 11, pp. 2987–3002, Nov 2019.
- [4] Z. Ba, S. Piao, X. Fu, D. Koutsonikolas, A. Mohaisen, and K. Ren, “Abc: Enabling smartphone authentication with built-in camera,” in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18 - 21, 2018*, 2018. [Online]. Available: <https://doi.org/10.14722/ndss.2018.23099>
- [5] C. Bo, L. Zhang, X.-Y. Li, Q. Huang, and Y. Wang, “Silentsense: Silent user identification via touch and movement behavioral biometrics,” in *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, ser. MobiCom ’13. New York, NY, USA: ACM, 2013, pp. 187–190. [Online]. Available: <http://doi.acm.org/10.1145/2500423.2504572>
- [6] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, “Mobile device identification via sensor fingerprinting,” *CoRR*, vol. abs/1408.1416, 2014. [Online]. Available: <http://arxiv.org/abs/1408.1416>
- [7] N. Bonettini, L. Bondi, D. Güera, S. Mandelli, P. Bestagini, S. Tubaro, and E. J. Delp, “Fooling prnu-based detectors through convolutional neural networks,” in *2018 26th European Signal Processing Conference (EUSIPCO)*, Sep. 2018, pp. 957–961.
- [8] C. Chen, W. Huang, B. Zhou, C. Liu, and W. H. Mow, “Picode: A new picture-embedding 2d barcode,” *IEEE Transactions on Image Processing*, vol. 25, no. 8, pp. 3444–3458, Aug 2016.
- [9] M. Chen, J. J. Fridrich, M. Goljan, and J. Lukás, “Determining image origin and integrity using sensor noise,” *IEEE Trans. Information Forensics and Security*, vol. 3, no. 1, pp. 74–90, 2008.
- [10] V. Christlein, C. Riess, J. Jordan, C. Riess, and E. Angelopoulou, “An evaluation of popular copy-move forgery detection approaches,” *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 6, pp. 1841–1854, Dec 2012.
- [11] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter, and G. Boato, “Raise: A raw images dataset for digital image forensics,” in *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 2015, pp. 219–224.
- [12] A. Das, N. Borisov, and M. Caesar, “Do you hear what I hear?: Fingerprinting smart devices through embedded acoustic components,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014, pp. 441–452. [Online]. Available: <http://doi.acm.org/10.1145/2660267.2660325>
- [13] —, “Tracking mobile web users through motion sensors: Attacks and defenses,” in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.
- [14] P. Eckersley, “How unique is your web browser?” in *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, ser. Lecture Notes in Computer Science, M. J. Atallah and N. J. Hopper, Eds., vol. 6205. Springer, 2010, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-642-14527-8_1
- [15] J. Entrieri and M. Kirchner, “Patch-based desynchronization of digital camera sensor fingerprints,” in *Media Watermarking, Security, and Forensics 2016, San Francisco, California, USA, February 14-18, 2016*, 2016, pp. 1–9. [Online]. Available: <http://ist.publisher.ingentaconnect.com/contentone/ist/ei/2016/00002016/00000008/art00022>

- [16] M. Everts, J.-H. Hoepman, and J. Siljee, “Ubikima: Ubiquitous authentication using a smartphone, migrating from passwords to strong cryptography,” in *Proceedings of the 2013 ACM Workshop on Digital Identity Management*, ser. DIM ’13. New York, NY, USA: ACM, 2013, pp. 19–24. [Online]. Available: <http://doi.acm.org/10.1145/2517881.2517885>
- [17] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song, “Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication,” *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 1, pp. 136–148, Jan 2013.
- [18] T. Gloe and R. Böhme, “The ‘Dresden Image Database’ for benchmarking digital image forensics,” in *Proceedings of the 25th Symposium On Applied Computing (ACM SAC 2010)*, vol. 2, 2010, pp. 1585–1591.
- [19] M. Goljan and J. Fridrich, “Determining approximate age of digital images using sensor defects,” in *SPIE Conference on Media Watermarking, Security, and Forensics*, 2011.
- [20] M. Goljan, J. Fridrich, and M. Chen, “Sensor noise camera identification: countering counter-forensics,” vol. 7541, 2010, pp. 75 410S–75 410S–12. [Online]. Available: <http://dx.doi.org/10.1117/12.839055>
- [21] M. Goljan, J. Fridrich, and T. Filler, “Large scale test of sensor fingerprint camera identification,” in *IS&T/SPIE Electronic Imaging*. International Society for Optics and Photonics, 2009, pp. 72 540I–72 540I.
- [22] M. Goljan and J. J. Fridrich, “Camera identification from cropped and scaled images,” in *Security, Forensics, Steganography, and Watermarking of Multimedia Contents X, San Jose, CA, USA, January 27, 2008*, ser. SPIE Proceedings, E. J. D. III, P. W. Wong, J. Dittmann, and N. D. Memon, Eds., vol. 6819. SPIE, 2008, p. 68190E.
- [23] M. Goljan, J. J. Fridrich, and M. Chen, “Defending against fingerprint-copy attack in sensor-based camera identification,” *IEEE Trans. Information Forensics and Security*, vol. 6, no. 1, pp. 227–236, March 2011. [Online]. Available: <https://doi.org/10.1109/TIFS.2010.2099220>
- [24] M. Goljan, J. J. Fridrich, and T. Filler, “Large scale test of sensor fingerprint camera identification,” in *Media Forensics and Security I, part of the IS&T-SPIE Electronic Imaging Symposium, San Jose, CA, USA, January 19-21, 2009, Proceedings*, 2009, p. 72540I. [Online]. Available: <https://doi.org/10.1117/12.805701>
- [25] N. Z. Gong, M. Payer, R. Moazzezi, and M. Frank, “Forgery-resistant touch-based authentication on mobile devices,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 499–510.
- [26] H. Gueddah, A. Yousfi, and M. Belkasmı, “The filtered combination of the weighted edit distance and the jaro-winkler distance to improve spellchecking arabic texts,” in *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, Nov 2015, pp. 1–6.
- [27] V. Hupert, D. Maier, and T. Müller, “Paying the price for disruption: How a fintech allowed account takeover,” in *ROOTS*, 2017.
- [28] V. Hupert, D. Maier, N. Schneider, J. Kirsch, and T. Müller, “Honey, I shrunk your app security: The state of android app hardening,” in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, ser. Lecture Notes in Computer Science, C. Giuffrida, S. Bardin, and G. Blanc, Eds., vol. 10885. Springer, 2018, pp. 69–91. [Online]. Available: https://doi.org/10.1007/978-3-319-93411-2_4
- [29] V. Hupert and T. Müller, “(in)security of app-based TAN methods in online banking,” Freidrich-Alexander-Universität Erlangen-Nürnberg, Tech. Rep., December 2015. [Online]. Available: <https://www1.cs.fau.de/content/insecurity-app-based-tan-methods-online-banking>
- [30] —, “On app-based matrix code authentication in online banking,” Freidrich-Alexander-Universität Erlangen-Nürnberg, Tech. Rep., 2016.
- [31] T. Hupperich, H. Hosseini, and T. Holz, *Leveraging Sensor Fingerprinting for Mobile Device Authentication*. Cham: Springer International Publishing, 2016, pp. 377–396. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40667-1_19
- [32] T. Hupperich, D. Maiorca, M. Kühner, T. Holz, and G. Giacinto, “On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms?” in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 191–200. [Online]. Available: <http://doi.acm.org/10.1145/2818000.2818032>
- [33] A. Karaküçük and A. E. Dirik, “Adaptive photo-response non-uniformity noise removal against image

source attribution,” *Digital Investigation: The International Journal of Digital Forensics and Incident Response*, vol. 12, 03 2015.

- [34] Y. Kim and Y. Lee, “Campuf: Physically unclonable function based on cmos image sensor fixed pattern noise,” in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3195970.3196005>
- [35] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. Freiling, “Fingerprinting mobile devices using personalized configurations,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 4–19, 2016.
- [36] C. T. Li and R. Satta, “On the location-dependent quality of the sensor pattern noise and its implication in multimedia forensics,” in *4th International Conference on Imaging for Crime Detection and Prevention 2011 (ICDP 2011)*, Nov 2011, pp. 1–6.
- [37] H. Li, W. Luo, Q. Rao, and J. Huang, “Anti-forensics of camera identification and the triangle test by improved fingerprint-copy attack,” *CoRR*, vol. abs/1707.07795, 2017. [Online]. Available: <http://arxiv.org/abs/1707.07795>
- [38] J. Lukás, J. Fridrich, and M. Goljan, “Digital camera identification from sensor pattern noise,” *IEEE Transactions on Information Forensics and Security*, vol. 1, no. 2, pp. 205–214, June 2006.
- [39] D. Maier, M. Protsenko, and T. Müller, “A game of droid and mouse: The threat of split-personality malware on Android,” *Computers & Security*, vol. 54, pp. 2–15, 2015. [Online]. Available: <http://dx.doi.org/10.1016/j.cose.2015.05.001>
- [40] F. Marra, F. Roli, D. Cozzolino, C. Sansone, and L. Verdoliva, “Attacking the triangle test in sensor-based camera identification,” in *2014 IEEE International Conference on Image Processing (ICIP)*, Oct 2014, pp. 5307–5311.
- [41] J. M. McCune, A. Perrig, and M. K. Reiter, “Seeing-is-believing: using camera phones for human-verifiable authentication,” in *2005 IEEE Symposium on Security and Privacy (S P'05)*, May 2005, pp. 110–124.
- [42] J. Millican and F. Stajano, *SAVVIcode: Preventing Mafia Attacks on Visual Code Authentication Schemes (Short Paper)*. Cham: Springer International Publishing, 2015, pp. 146–152. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24192-0_10
- [43] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert, “SMS-based one-time passwords: attacks and defense,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Berlin Heidelberg, 2013, pp. 150–159.
- [44] A. Naveh and E. Tromer, “Photoproof: Cryptographic image authentication for any set of permissible transformations,” in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 255–271.
- [45] C. Nickel, T. Wirtl, and C. Busch, “Authentication of smartphone users based on the way they walk using k-NN algorithm,” in *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, July 2012, pp. 16–20.
- [46] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless monster: Exploring the ecosystem of web-based device fingerprinting,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 541–555. [Online]. Available: <https://doi.org/10.1109/SP.2013.43>
- [47] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, “TESSERACT: Eliminating experimental bias in malware classification across space and time,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 729–746. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury>
- [48] E. Quiring and M. Kirchner, “Fragile sensor fingerprint camera identification,” in *2015 IEEE International Workshop on Information Forensics and Security (WIFS)*, Nov 2015, pp. 1–6.
- [49] E. Quiring, M. Kirchner, and K. Rieck, “On the security and applicability of fragile camera fingerprints,” in *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., vol. 11735. Springer, 2019, pp. 450–470. [Online]. Available: https://doi.org/10.1007/978-3-030-29959-0_22
- [50] R. Ramanath, W. E. Snyder, Y. Yoo, and M. S. Drew, “Color image processing pipeline,” *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 34–43, Jan 2005.
- [51] B. Reaves, L. Blue, H. Abdullah, L. Vargas, P. Traynor, and T. Shrimpton, “Authenticall: Efficient identity and content authentication for phone calls,” in *26th USENIX Security Symposium, USENIX*

Security 2017, Vancouver, BC, Canada, August 16-18, 2017., 2017, pp. 575–592. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/reaves>

- [52] D. Shullani, M. Fontani, M. Iuliani, O. Al Shaya, and A. Piva, “Vision: a video and image dataset for source identification,” *EURASIP Journal on Information Security*, vol. 2017, no. 1, p. 15, 2017.
- [53] M. Simkin, D. Schröder, A. Bulling, and M. Fritz, *Ubic: Bridging the Gap between Digital Cryptography and the Physical World*. Cham: Springer International Publishing, 2014, pp. 56–75. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11203-9_4
- [54] T. Stöber, M. Frank, J. B. Schmitt, and I. Martinovic, “Who do you sync you are?: smartphone fingerprinting via application behaviour,” in *Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC’13, Budapest, Hungary, April 17-19, 2013*, ser. WiSec ’13. New York, NY, USA: ACM, 2013, pp. 7–12. [Online]. Available: <http://doi.acm.org/10.1145/2462096.2462099>
- [55] S. Taspinar, M. Mohanty, and N. Memon, “PRNU based source attribution with a collection of seam-carved images,” in *2016 IEEE International Conference on Image Processing (ICIP)*, Sept 2016, pp. 156–160.
- [56] D. Valsesia, G. Coluccia, T. Bianchi, and E. Magli, “User authentication via prnu-based physical unclonable functions,” *IEEE Trans. Information Forensics and Security*, vol. 12, no. 8, pp. 1941–1956, 2017. [Online]. Available: <https://doi.org/10.1109/TIFS.2017.2697402>
- [57] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, “FP-STALKER: tracking browser fingerprint evolutions,” in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 728–741. [Online]. Available: <https://doi.org/10.1109/SP.2018.00008>
- [58] B. Wronski, I. Garcia-Dorado, M. Ernst, D. Kelly, M. Krainin, C.-K. Liang, M. Levoy, and P. Milanfar, “Handheld multi-frame super-resolution,” *ACM Trans. Graph.*, vol. 38, no. 4, pp. 28:1–28:18, Jul. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3306346.3323024>
- [59] W. Wu, J. Wu, Y. Wang, Z. Ling, and M. Yang, “Efficient fingerprinting-based android device identification with zero-permission identifiers,” *IEEE Access*, vol. 4, pp. 8073–8083, 2016. [Online]. Available: <https://doi.org/10.1109/ACCESS.2016.2626395>

- [60] C. Yue, “Sensor-based mobile web fingerprinting and cross-site input inference attacks,” in *Security and Privacy Workshops (SPW), 2016 IEEE*. IEEE, 2016, pp. 241–244.
- [61] C. Zhang and H. Zhang, “Exposing digital image forgeries by using canonical correlation analysis,” in *Pattern Recognition (ICPR), 2010 20th International Conference on*, Aug 2010, pp. 838–841.
- [62] Z. Zhou, W. Diao, X. Liu, and K. Zhang, “Acoustic fingerprinting revisited: Generate stable device ID stealthily with inaudible sound,” *CoRR*, vol. abs/1407.0803, 2014. [Online]. Available: <http://arxiv.org/abs/1407.0803>

A The PRNU

In this part of the Appendix, we derive the mathematical foundation of PRNU, as it is well established in literature, most prominently in [38] and [21]. Furthermore, we explain the standard way to compare two noise patterns to determine if they were made by the same camera.

The image I , read out at the camera-interface, differs from the perfect ideal image I_0 that might exist at the sensor without any imperfections or perturbations. First i.i.d. sources like shot-noise or dark-current-noise and random-noise are additionally in the image. These noises are summarized in θ . Then, the described PRNU as multiplicative source of noise is casted on the image, expressed in the following equation with K . The imaging formation pipeline catering for those additional corruptions is given by

$$I = I_0 + I_0K + \theta . \quad (1)$$

The mentioned additive noise sources θ can be averaged out over multiple images. However, the multiplicative noise K , the PRNU we are seeking for, is not a random noise for a given pixel at position (x, y) . By averaging a number of photos and filtering them to reduce other noise, the PRNU can be extracted.

The procedure introduced by Lukás *et al.* [38] starts with suppressing the image content of the single images, by taking the difference of the low-pass filtered version of the image and the image itself. This is described with

$$W_I = I - F(I) , \quad (2)$$

where F could any arbitrary low-pass filter, in practice, however usually a wavelet filter is used.

The residuals of this filter operation are weighted by the pixel intensities and averaged over N samples like

$$\hat{K} = \frac{\sum_{i=1}^N W_I^i I^i}{\sum_{i=1}^N (I^i)^2} . \quad (3)$$

The superscript i in this equation is an image index going from 1 to N . Also it is worth pointing out, that the product

W^i and similar operations are element-wise, e.g. at pixel positions (x,y) and not regular matrix multiplications. Finally, we are left with the estimated PRNU fingerprint \hat{K} .

B HR-QR Implementation

The apps developed for this paper also include a practical implementation of the HR-QR scheme. For the actual implementation of an human-readable QR code, we needed to have an error resilient solution. OCR results are never perfect and other values may falsely be factored into the comparison. With the knowledge that this allows for (easier to spot) attacks by exchanging single numbers or letters, we suggest the use of the *Jaro-Winkler distance* D_{jw} , as a metric to check string similarities [26]. Given the strings of the text X and a string of the QR Code Y , the metric is calculated with

$$D_{jw}(X, Y) = D_j(X, Y) + l \cdot p \cdot (1 - D_j(X, Y)) \quad , \quad (5)$$

where p is a prefix coefficient, which promotes the chains with the longest common prefix, and l being the common prefix between X and Y . We use the default value of 0.1 for p . Further, the distances in D_j are calculated by

$$D_j(X, Y) = \frac{1}{3} \left(\frac{c}{|X|} + \frac{c}{|Y|} + \frac{c-t}{c} \right) \quad , \quad (6)$$

with c the number of characters that match between X and Y and t the number of transpositions.

As more text is potentially in the users' view, for our proof of concept we extracted the distance for substrings, as shown in algorithm 1. A future implementation might instead refrain to an algorithm already fit for substring matching, like an *Smith-Waterman algorithm* or use a way to transmit the human-readable part intact while still being easy to read, similar to the proposals of Millican and Stajano [42] and Simkin *et al.* [53].

Algorithm 1 MaxJaro

```

1: procedure MAXJARO(check, text)
2:   maxSimilarity ← JARONWINKLERDIS-
   TANCE(check, text)
3:   for each substr of text where LEN(substr) =
   LEN(check) or LEN(check) + 1 do
4:     similarity ← JARONWINKLERDISTANCE(check,
   substr)
5:     maxSimilarity ← MAX(max, similarity)
6:   end for
7:   return maxSimilarity
8: end procedure

```

Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities

Manh-Dung Nguyen

Univ. Paris-Saclay, CEA LIST, France
manh-dung.nguyen@cea.fr

Sébastien Bardin

Univ. Paris-Saclay, CEA LIST, France
sebastien.bardin@cea.fr

Richard Bonichon

Tweag I/O, France
richard.bonichon@tweag.io

Roland Groz

Univ. Grenoble Alpes, France
roland.groz@univ-grenoble-alpes.fr

Matthieu Lemerre

Univ. Paris-Saclay, CEA LIST, France
matthieu.lemerre@cea.fr

Abstract

Directed fuzzing focuses on automatically testing specific parts of the code by taking advantage of additional information such as (partial) bug stack trace, patches or risky operations. Key applications include bug reproduction, patch testing and static analysis report verification. Although directed fuzzing has received a lot of attention recently, hard-to-detect vulnerabilities such as Use-After-Free (UAF) are still not well addressed, especially at the binary level. We propose UAFUZZ, the first (binary-level) directed greybox fuzzer dedicated to UAF bugs. The technique features a fuzzing engine tailored to UAF specifics, a lightweight code instrumentation and an efficient bug triage step. Experimental evaluation for bug reproduction on real cases demonstrates that UAFUZZ significantly outperforms state-of-the-art directed fuzzers in terms of fault detection rate, time to exposure and bug triaging. UAFUZZ has also been proven effective in patch testing, leading to the discovery of 30 new bugs (7 CVEs) in programs such as Perl, GPAC and GNU Patch. Finally, we provide to the community a large fuzzing benchmark dedicated to UAF, built on both real codes and real bugs.

1 Introduction

Context. Finding bugs early is crucial in the vulnerability management process. The recent rise of fuzzing [50, 53] in both academia and industry, such as Microsoft’s Springfield [52] and Google’s OSS-FUZZ [14], shows its ability to find various types of bugs in real-world applications. *Coverage-based Greybox Fuzzing* (CGF), such as AFL [1] and LIBFUZZER [13], leverages code coverage information in order to guide input generation toward new parts of the program under test (PUT), exploring as many program states as possible in the hope of triggering crashes. On the other hand, *Directed Greybox Fuzzing* (DGF) [25, 28] aims to perform stress testing on pre-selected potentially vulnerable target locations, with applications to different security contexts: (1) *bug reproduction* [25, 28, 42, 61], (2) *patch testing* [25, 51, 59] or (3) *static analysis report verification* [31, 49]. Depending on the application, target locations are originated from *bug stack traces*, patches or static analysis reports.

We focus mainly on *bug reproduction*, which is the most common practical application of DGF [25, 28, 42, 49, 74]. It consists in generating Proof-of-Concept (PoC) inputs of disclosed vulnerabilities given bug report information. It is especially needed since only 54.9% of usual bug reports can

be reproduced due to missing information and users’ privacy violation [54]. Even with a PoC provided in the bug report, developers may still need to consider all corner cases of the bug in order to avoid regression bugs or incomplete fixes. In this case, providing more bug-triggering inputs becomes important to facilitate and accelerate the repair process. *Bug stack traces*, sequences of function calls at the time a bug is triggered, are widely used for guiding directed fuzzers [25, 28, 42, 49]. Running a code on a PoC input under profiling tools like AddressSanitizer (ASan) [65] or VALGRIND [55] will output such a bug stack trace.

Problem. Despite tremendous progress in the past few years [5, 21, 23, 25, 28, 29, 39, 46, 47, 60, 62, 67, 73], current (directed or not) greybox fuzzers still have a hard time finding *complex vulnerabilities* such as Use-After-Free (UAF), non-interference or flaky bugs [24], which require bug-triggering paths satisfying very specific properties. For example, OSS-FUZZ [14, 15] or recent greybox fuzzers [25, 62, 73] only found a small number of UAF. Actually, RODEODAY [16], a continuous bug finding competition, recognizes that fuzzers should aim to cover new bug classes like UAF in the future [37], moving further from the widely-used LAVA [36] bug corpora which only contains buffer overflows.

We focus on UAF bugs. They appear when a heap element is used after having been freed. The numbers of UAF bugs has increased in the National Vulnerability Database (NVD) [20]. They are currently identified as *one of the most critical exploitable vulnerabilities* due to the lack of mitigation techniques compared to other types of bugs, and they may have serious consequences such as data corruption, information leaks and denial-of-service attacks.

Goal and challenges. *We focus on the problem of designing an efficient directed fuzzing method tailored for UAF.* The technique must also be able to work at binary-level (no source-level instrumentation), as source codes of security-critical programs are not always available or may rely partly on third-party libraries. However, fuzzers targeting the detection of UAF bugs confront themselves with the following challenges.

- C1. Complexity** – Exercising UAF bugs require to generate inputs triggering a *sequence* of 3 events – *alloc*, *free* and *use* – *on the same memory location*, spanning multiple functions of the PUT, where buffer overflows only require a single out-of-bound memory access. This combination of both *temporal* and *spatial* constraints is extremely difficult to meet in practice;
- C2. Silence** – UAF bugs often have *no observable effect*,

Table 1: Summary of existing greybox fuzzing techniques.

	AFL	AFLGO	HAWKEYE	UAFUZZ
Directed fuzzing approach	✗	✓	✓	✓
Support binary	✓	✗	✗	✓
UAF bugs oriented	✗	✗	✗	✓
Fast instrumentation	✓	✗	✗	✓
UAF bugs triage	✗	✗	✗	✓

such as segmentation faults. In this case, fuzzers simply observing crashing behaviors do not detect that a test case triggered such a memory bug. Sadly, popular profiling tools such as ASan or VALGRIND cannot be used in a fuzzing context due to their high runtime overhead.

Actually, current state-of-the-art directed fuzzers, namely AFLGO [25] and HAWKEYE [28], fail to address these challenges. First, they are too generic and therefore do not cope with the specificities of UAF such as temporality – their guidance metrics do not consider any notion of sequenceness. Second, they are completely blind to UAF bugs, requiring to send all the many generated seeds to a profiling tool for an expensive extra check. Finally, current implementations of source-based DGF fuzzers typically suffer from an expensive instrumentation step [3], e.g., AFLGO spent nearly 2h compiling and instrumenting `cxxfilt` (Binutils).

Proposal. We propose UAFUZZ, the first (binary-level) directed greybox fuzzer tailored to UAF bugs. A quick comparison of UAFUZZ with existing greybox fuzzers in terms of UAF is presented in Table 1. While we follow mostly the generic scheme of directed fuzzing, we carefully tune several of its key components to the specifics of UAF:

- the *distance metric* favors shorter call chains leading to the target functions that are more likely to include both allocation and free functions – where sota directed fuzzers rely on a generic CFG-based distance;
- *seed selection* is now based on a *sequenceness-aware target similarity metric* – where sota directed fuzzers rely at best on target coverage;
- our *power schedule* benefits from these new metrics, plus another one called *cut-edges* favoring prefix paths more likely to reach the whole target.

Finally, the bug triaging step piggy-backs on our previous metrics to pre-identifies seeds as likely-bugs or not, sparing a huge amount of queries to the profiling tool for confirmation (VALGRIND in our implementation).

Contributions. Our contribution is the following:

- We design the first directed greybox fuzzing technique tailored to UAF bugs (Section 4). Especially, we systematically revisit the three main ingredients of directed fuzzing (selection heuristic, power schedule, input metrics) and specialize them to UAF. These improvements are proven beneficial and complementary;
- We develop a toolchain [19] on top of the state-of-the-art greybox fuzzer AFL [1] and the binary analysis platform BINSEC [4], named UAFUZZ, implementing the above method for UAF directed fuzzing over binary

codes (Section 5) and enjoying small overhead;

- We construct and openly release [18] the largest fuzzing benchmark dedicated to UAF, comprising 30 real bugs from 17 widely-used projects (including the few previous UAF bugs found by directed fuzzers), in the hope of facilitating future UAF fuzzing evaluation;
- We evaluate our technique and tool in a bug reproduction setting (Section 6), demonstrating that UAFUZZ is highly effective and significantly outperforms state-of-the-art competitors: 2× faster in average to trigger bugs (up to 43×), +34% more successful runs in average (up to +300%) and 17× faster in triaging bugs (up to 130×, with 99% spare checks);
- Finally, UAFUZZ is also proven effective in patch testing (§6.7), leading to the discovery of 30 unknown bugs (11 UAFs, 7 CVEs) in projects like Perl, GPAC, MuPDF and GNU Patch (including 4 buggy patches). So far, 17 have been fixed.

UAFUZZ is the *first* directed greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only bug stack traces. UAFUZZ outperforms existing directed fuzzers on this class of vulnerabilities for bug reproduction and encouraging results have been obtained as well on patch testing. We believe that our approach may also be useful in slightly related contexts, for example partial bug reports from static analysis or other classes of vulnerabilities.

2 Background

Let us first clarify some notions used along the paper.

2.1 Use-After-Free

Execution. An *execution* is the complete sequence of states executed by the program on an *input*. An execution trace *crashes* when it ends with a visible error. The standard goal of fuzzers is to find inputs leading to crashes, as crashes are the first step toward exploitable vulnerabilities.

UAF bugs. *Use-After-Free* (UAF) bugs happen when dereferencing a pointer to a heap-allocated object which is no longer valid (i.e., the pointer is *dangling*). Note that *Double-Free* (DF) is a special case.

UAF-triggering conditions. Triggering a UAF bug requires to find an input whose execution covers in sequence *three* UAF events: an allocation (*alloc*), a *free* and a *use* (typically, a dereference), *all three referring to the same memory object*, as shown in Listing 1.

```

1 char *buf = (char *) malloc(BUF_SIZE);
2 free(buf); // pointer buf becomes dangling
3 ...
4 strncpy(buf, argv[1], BUF_SIZE-1); // Use-After-Free

```

Listing 1: Code snippet illustrating a UAF bug.

Furthermore, this last *use* generally does not make the execution immediately crash, as a memory violation crashes a

process only when it accesses an address outside of the address space of the process, which is unlikely with a dangling pointer. Thus, UAF bugs go often unnoticed and are a good vector of exploitation [45, 75].

2.2 Stack Traces and Bug Traces

By inspection of the state of a process we can extract a *stack trace*, i.e. the list of the function calls active in that state. Stack traces are easily obtained from a process when it crashes. As they provide (partial) information about the sequence of program locations leading to a crash, they are extremely valuable for bug reproduction [25, 28, 42, 49].

Yet, as crashes caused by UAF bugs may happen long after the UAF happened, standard stack traces usually do not help in reproducing UAF bugs. Hopefully, profiling tools for dynamically detecting memory corruptions, such as ASan [65] or VALGRIND [55], record the stack traces of *all memory-related events*: when they detect that an object is used after being freed, they actually report *three stack traces* (when the object is allocated, when it is freed and when it is used after being freed). We call such a sequence of three stack traces a **(UAF) bug trace**. When we use a bug trace as an input to try to reproduce the bug, we call such a bug trace a *target*.

```
// stack trace for the bad Use
==4440== Invalid read of size 1
==4440== at 0x40A8383: vfprintf (vfprintf.c:1632)
==4440== by 0x40A8670: buffered_vfprintf (vfprintf.c:2320)
==4440== by 0x40A62D0: vfprintf (vfprintf.c:1293)
==4440== by 0x80AA58A: error (elfcomm.c:43)
==4440== by 0x8085384: process_archive (readelf.c:19063)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)

// stack trace for the Free
==4440== Address 0x421fdc8 is 0 bytes inside a block of size 86 free'd
==4440== at 0x402D358: free (in vgpreload_memcheck-x86-linux.so)
==4440== by 0x80857B4: process_archive (readelf.c:19178)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)

// stack trace for the Alloc
==4440== Block was alloc'd at
==4440== at 0x402C17C: malloc (in vgpreload_memcheck-x86-linux.so)
==4440== by 0x80AC687: make_qualified_name (elfcomm.c:906)
==4440== by 0x80854BD: process_archive (readelf.c:19089)
==4440== by 0x8085A57: process_file (readelf.c:19242)
==4440== by 0x8085C6E: main (readelf.c:19318)
```

Figure 1: Bug trace of CVE-2018-20623 (UAF) produced by VALGRIND.

2.3 Directed Greybox Fuzzing

Fuzzing [50, 53] consists in stressing a code under test through massive input generation in order to find bugs. Recent *coverage-based greybox fuzzers* (CGF) [1, 13] rely on *lightweight* program analysis to guide the search – typically through coverage-based feedback. Roughly speaking, a *seed* (input) is *avored* (selected) when it reaches under-explored parts of the code, and such favored seeds are then *mutated* to create new seeds for the code to be executed on. CGF is geared toward covering code in the large, in the hope of finding unknown vulnerabilities.

On the other hand, *directed greybox fuzzing* (DGF) [25, 28] aims at reaching a *pre-identified potentially buggy part of*

the code from a *target* (e.g., patch, static analysis report), as often and fast as possible. Directed fuzzers follow the general principles and architecture as CGF, but adapt the key components to their goal, essentially favoring seeds “*closer*” to the target. Overall directed fuzzers¹ are built upon three main steps: (1) *instrumentation* (distance pre-computation), (2) *fuzzing* (including seed selection, power schedule and seed mutation) and (3) *triage*.

The standard core algorithm of DGF is presented in Algorithm 1 (the parts we modify in UAFUZZ are in gray). Given a program P , a set of initial seeds S_0 and a target T , the algorithm outputs a set of bug-triggering inputs S' . The fuzzing queue S is initialized with the initial seeds in S_0 (line 1).

1. DGF first performs a static analysis (e.g., *target distance computation* for each basic block) and insert the instrumentation for dynamic coverage or distance information (line 2);
2. The fuzzer then repeatedly mutates inputs s chosen from the fuzzing queue S (line 4) until a timeout is reached. An input is selected either if it is *favored* (i.e., believed to be interesting) or with a small probability α (line 5). Subsequently, DGF assigns the *energy* (a.k.a, the number M of mutants to be created) to the selected seed s (line 6) and monitors their executions (line 9). If the generated mutant s' crashes the program, it is added to the set S' of crashing inputs (line 11). Also, newly generated mutants are added to the fuzzing queue² (line 13);
3. Finally, DGF returns S' as the set of bug-triggering inputs (triage does nothing in standard DGF) (line 14).

Algorithm 1: Directed Greybox Fuzzing

Input : Program P ; Initial seeds S_0 ; Target locations T

Output : Bug-triggering seeds S'

```
1  $S' := \emptyset$ ;  $S := S_0$ ; ▷  $S$ : the fuzzing queue
2  $P' \leftarrow \text{PREPROCESS}(P, T)$ ; ▷ phase 1: Instrumentation
3 while timeout not exceeded do ▷ phase 2: Fuzzing
4   for  $s \in S$  do
5     if  $\text{IS\_FAVORED}(s)$  or  $\text{rand}() \leq \alpha$  then
6       ▷ seed selection,  $\alpha$ : small probability
7        $M := \text{ASSIGN\_ENERGY}(s)$ ; ▷ power schedule
8       for  $i \in 1 \dots M$  do
9          $s' := \text{mutate\_input}(s)$ ; ▷ seed mutation
10         $\text{res} := \text{run}(P', s', T)$ ;
11        if  $\text{is\_crash}(\text{res})$  then
12           $S' := S' \cup \{s'\}$ ; ▷ crashing inputs
13        else
14           $S := S \cup \{s'\}$ ;
14  $S' = \text{TRIAGE}(S, S')$ ; ▷ phase 3: Triage
15 return  $S'$ ;
```

¹And coverage-based fuzzers.

²This is a general view. In practice, seeds regarded as very uninteresting are already discarded at this point.

AFLGo [25] was the first to propose a CFG-based distance to evaluate the proximity between a seed execution and multiple targets, together with a simulated annealing-based power schedule. HAWKEYE [28] keeps the CFG-based view but improves its accuracy³, brings a seed selection heuristic partly based on target coverage (seen as a set of locations) and proposes adaptive mutations.

3 Motivation

The toy example in Listing 2 contains a UAF bug due to a missing `exit()` call, a common root cause in such bugs (e.g., CVE-2014-9296, CVE-2015-7199). The program reads a file and copies its contents into a buffer `buf`. Specifically, a memory chunk pointed at by `p` is allocated (line 12), then `p_alias` and `p` become aliased (line 15). The memory pointed by both pointers is freed in function `bad_func` (line 10). The UAF bug occurs when the freed memory is dereferenced again via `p` (line 19).

Bug-triggering conditions. The UAF bug is triggered iff the first three bytes of the input are ‘AFU’. To quickly detect this bug, fuzzers need to explore the right path through the `if` part of conditional statements in lines 13, 5 and 18 in order to cover in sequence the three UAF events *alloc*, *free* and *use* respectively. It is worth noting that this UAF bug does not make the program crash, hence existing greybox fuzzers without sanitization will not detect this memory error.

Coverage-based Greybox Fuzzing. Starting with an empty seed, AFL quickly generates 3 new inputs (e.g., ‘AAAA’, ‘FFFF’ and ‘UUUU’) triggering individually the 3 UAF events. None of these seeds triggers the bug. As the probability of generating an input starting with ‘AFU’ from an empty seed is extremely small, the coverage-guided mechanism is not effective here in tracking a sequence of UAF events even though each individual event is easily triggered.

Directed Greybox Fuzzing. Given a bug trace (14 – *alloc*, 17, 6, 3 – *free*, 19 – *use*) generated for example by ASan, DGF prevents the fuzzer from exploring undesirable paths, e.g., the `else` part at line 7 in function `func`, in case the condition at line 5 is more complex. Still, directed fuzzers have their own blind spots. For example, standard DGF seed selection mechanisms favor a seed whose execution trace covers many locations in targets, instead of trying to reach these locations in a given order. For example, regarding a target (A, F, U), standard DGF distances [25, 28] do not discriminate between an input s_1 covering a path $A \rightarrow F \rightarrow U$ and another input s_2 covering $U \rightarrow A \rightarrow F$. The lack of ordering in exploring target locations makes UAF bug detection very challenging for existing directed fuzzers. Another example: the power function proposed by HAWKEYE [28] may assign much energy to a seed whose trace does not reach the target function, implying that it could get lost on the toy example in the `else` part at line 7 in function `func`.

³Possibly at the price of both higher pre-computation costs due to more precise static analysis and runtime overhead due to complex seed metrics.

```

1 int *p, *p_alias;
2 char buf[10];
3 void bad_func(int *p) {free(p);} /* exit() is missing */
4 void func() {
5     if (buf[1] == 'F')
6         bad_func(p);
7     else /* lots more code ... */
8 }
9 int main (int argc, char *argv[]) {
10     int f = open(argv[1], O_RDONLY);
11     read(f, buf, 10);
12     p = malloc(sizeof(int));
13     if (buf[0] == 'A'){
14         p_alias = malloc(sizeof(int));
15         p = p_alias;
16     }
17     func();
18     if (buf[2] == 'U')
19         *p = 1;
20     return 0;
21 }

```

Listing 2: Motivating example.

A glimpse at UAFUZZ. We rely in particular on modifying the seed selection heuristic w.r.t. the number of targets covered by an execution trace (§4.2) and bringing target ordering-aware seed metrics to DGF (§4.3).

On the toy example, UAFUZZ generates inputs to progress towards the expected target sequences. For example, in the same fuzzing queue containing 4 inputs, the mutant ‘AFAA’, generated by mutating the seed ‘AAAA’, is discarded by AFL as it does not increase code coverage. However, since it has maximum value of target similarity metric score (i.e., 4 targets including lines 14, 17, 6, 3) compared to all 4 previous inputs in the queue (their scores are 0 or 2), this mutant is selected by UAFUZZ for subsequent fuzzing campaigns. By continuing to fuzz ‘AFAA’, UAFUZZ eventually produces a bug-triggering input, e.g., ‘AFUA’.

Evaluation. AFLGo (source-level) cannot detect the UAF bug within 2 hours⁴⁵, while UAFUZZ (binary-level) is able to trigger it within 20 minutes. Also, UAFUZZ sends a single input to VALGRIND for confirmation (the right PoC input), while AFLGo sends 120 inputs.

4 The UAFUZZ Approach

UAFUZZ is made out of several components encompassing seed selection (§4.2), input metrics (§4.3), power schedule (§4.4), and seed triage (§4.5). Before detailing these aspects, let us start with an overview of the approach.

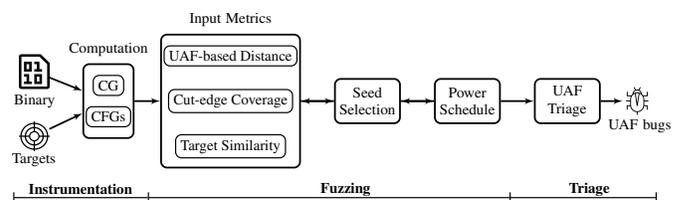


Figure 2: Overview of UAFUZZ.

⁴AFL-QEMU did not succeed either.

⁵HAWKEYE is not available and thus could not be tested.

We aim to find an input fulfilling both control-flow (temporal) and runtime (spatial) conditions to trigger the UAF bug. We solve this problem by bringing UAF characteristics into DGF in order to generate more potential inputs reaching targets in sequence w.r.t. the UAF expected bug trace. Figure 2 depicts the general picture. Especially:

- We propose three dynamic seed metrics specialized for UAF vulnerabilities detection. The distance metric approximates how close a seed is to all target locations (§4.3), and takes into account the need for the seed execution trace to cover the three UAF events in order. The cut-edge coverage metric (§4.4.1) measures the ability of a seed to take the correct decision at important decision nodes. Finally, the target similarity metric concretely assesses how many targets a seed execution trace covers at runtime (§4.2.2);
- Our seed selection strategy (§4.2) favors seeds covering more targets at runtime. The power scheduler determining the energy for each selected seed based on its metric scores during the fuzzing process is detailed in §4.4;
- Finally, we take advantage of our previous metrics to pre-identify likely-PoC inputs that are sent to a profiling tool (here VALGRIND) for bug confirmation, avoiding many useless checks (§4.5).

4.1 Bug Trace Flattening

A bug trace (§2.2) is a sequence of stack traces, i.e. it is a large object not fit for the lightweight instrumentation required by greybox fuzzing. The most valuable information that we need to extract from a bug trace is the sequence of basic blocks (and functions) that were traversed, which is an easier object to work with. We call this extraction *bug trace flattening*.

The operation works as follows. First, each of the three stack-traces is seen as a path in a call tree; we thus merge all the stack traces to re-create that tree. Some of the nodes in the tree have several children; we make sure that the children are ordered according to the ordering of the UAF events (i.e. the child coming from the *alloc* event comes before the child coming from the *free* event). Figure 3 shows an example of tree for the bug trace given in Figure 1.

Finally, we perform a preorder traversal of this tree to get a

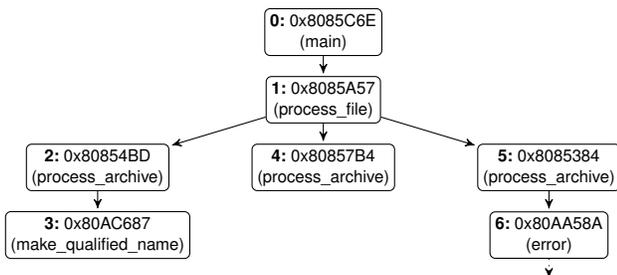


Figure 3: Reconstructed tree from CVE-2018-20623 (bug trace from Figure 1). The preorder traversal of this tree is simply $0 \rightarrow 1 \rightarrow 2 \rightarrow 3(n_{alloc}) \rightarrow 4(n_{free}) \rightarrow 5 \rightarrow 6(n_{use})$.

sequence of target locations (and their associated functions), which we will use in the following algorithms.

4.2 Seed Selection based on Target Similarity

Fuzzers generate a large number of inputs so that smartly selecting the seed from the fuzzing queue to be mutated in the next fuzzing campaign is crucial for effectiveness. Our seed selection algorithm is based on two insights. First, we *should prioritize seeds that are most similar to the target bug trace*, as the goal of a directed fuzzer is to find bugs covering the target bug trace. Second, *target similarity should take ordering (a.k.a. sequenceness) into account*, as traces covering sequentially a number of locations in the target bug trace are closer to the target than traces covering the same locations in an arbitrary order.

4.2.1 Seed Selection

Definition 1. A *max-reaching input* is an input s whose execution trace is the most similar to the target bug trace T so far, where most similar means “having the highest value as compared by a target similarity metric $t(s, T)$ ”.

Algorithm 2: IS_FAVORED

Input : A seed s
Output : *true* if s is favored, otherwise *false*

```

1 global  $t_{max} = 0$ ;           ▷ maximum target similar metric score
2 if  $t(s) \geq t_{max}$  then  $t_{max} = t(s)$ ; return true;           ▷ update  $t_{max}$ 
3 else if  $new\_cov(s)$  then return true;           ▷ increase coverage
4 else return false;

```

We mostly select and mutate max-reaching inputs during the fuzzing process. Nevertheless, we also need to improve code coverage, thus UAFUZZ also selects inputs that cover new paths, with a small probability α (Algorithm 1). In our experiments, the probability of selecting the remaining inputs in the fuzzing queue that are less favored is 1% like AFL [1].

4.2.2 Target Similarity Metrics

A *target similarity metric* $t(s, T)$ measures the similarity between the execution of a seed s and the target UAF bug trace T . We define 4 such metrics, based on whether we consider ordering of the covered targets in the bug trace (P), or not (B) – P stands for Prefix, B for Bag; and whether we consider the full trace, or only the three UAF events ($3T$):

- Target prefix $t_P(s, T)$: locations in T covered in sequence by executing s until first divergence;
- UAF prefix $t_{3TP}(s, T)$: UAF events of T covered in sequence by executing s until first divergence;
- Target bag $t_B(s, T)$: locations in T covered by executing s ;
- UAF bag $t_{3TB}(s, T)$: UAF events of T covered by s .

For example, using Listing 2, the 4 metric values of a seed s ‘ABUA’ w.r.t. the UAF bug trace T are:

$t_P(s, T) = 2$, $t_{3TP}(s, T) = 1$, $t_B(s, T) = 3$ and $t_{3TB}(s, T) = 2$.

These 4 metrics have different degrees of *precision*. A metric t is said *more precise than* a metric t' if, for any two

seeds s_1 and s_2 : $t(s_1, T) \geq t(s_2, T) \Rightarrow t'(s_1, T) \geq t'(s_2, T)$. Figure 4 compares our 4 metrics w.r.t their relative precision.

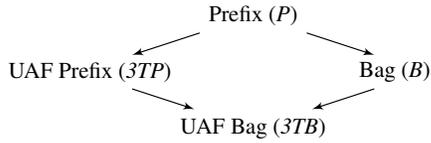


Figure 4: Precision lattice for Target Similarity Metrics

4.2.3 Combining Target Similarity Metrics

Using a precise metric such as P allows to better assess progression towards the goal. In particular, P can distinguish seeds that match the target bug trace from those that do not, while other metrics cannot. On the other hand, a less precise metric provides information that precise metrics do not have. For instance, P does not measure any difference between traces whose suffix would match the target bug trace, but who would diverge from the target trace on the first locations (like ‘UUU’ and ‘UFU’ on Listing 2), while B can.

To take benefit from both precise and imprecise metrics, we combine them using a lexicographical order. Hence, the P -3TP- B metric is defined as:

$$t_{P-3TP-B}(s, T) \triangleq \langle t_P(s, T), t_{3TP}(s, T), t_B(s, T) \rangle$$

This combination favors first seeds that cover the most locations in the prefix, then (in case of tie) those reaching the most number of UAF events in sequence, and finally (in case of tie) those that reach the most locations in the target. Based on preliminary investigation, we default to P -3TP- B for seed selection in UAFUZZ.

4.3 UAF-based Distance

One of the main component of directed greybox fuzzers is the computation of a *seed distance*, which is an evaluation of a distance from the execution trace of a seed s to the target. The main heuristic here is that if the execution trace of s is close to the target, then s is close to an input that would cover the target, which makes s an interesting seed. In existing directed greybox fuzzers [2, 28], the seed distance is computed to a target which is a single location or a set of locations. This is not appropriate for the reproduction of UAF bugs, that must go through 3 different locations in sequence. Thus, we propose to modify the seed distance computation to take into account the need to reach the locations in order.

4.3.1 Zoom: Background on Seed Distances

Existing directed greybox fuzzers [2, 28] compute the distance $d(s, T)$ from a seed s to a target T as follows.

AFLGO’s seed distance [2]. The *seed distance* $d(s, T)$ is defined as the (arithmetic) mean of the *basic-block distances* $d_b(m, T)$, for each basic block m in the execution trace of s .

The *basic-block distance* $d_b(m, T)$ is defined using the length of the intra-procedural shortest path from m to the basic block of a “call” instruction, using the CFG of the function containing m ; and the length of the inter-procedural shortest path from the function containing m to the target functions T_f (in our case, T_f is the function where the *use* event happens), using the call graph.

HAWKEYE’s enhancement [28]. The main factor in this seed distance computation is computing distance between functions in the call graph. To compute this, AFLGO uses the original call graph with every edge having weight 1. HAWKEYE improves this computation by proposing the augmented adjacent-function distance (AAFD), which changes the edge weight from a caller function f_a and a callee f_b to $w_{Hawkeye}(f_a, f_b)$. The idea is to favor edges in the call graph where the callee can be called in a variety of situations, i.e. appear several times at different locations.

4.3.2 Our UAF-based Seed Distance

Previous seed distances [2, 28] do not account for any order among the target locations, while it is essential for UAF. We address this issue by modifying the distance between functions in the call graph to favor paths that *sequentially* go through the three UAF events *alloc*, *free* and *use* of the bug trace. This is done by decreasing the weight of the edges in the call graph that are likely to be between these events, using lightweight static analysis.

This analysis first computes R_{alloc} , R_{free} , and R_{use} , i.e., the sets of functions that can call respectively the *alloc*, *free*, or *use* function in the bug trace – the *use* function is the one where the *use* event happens. Then, we consider each call edge between f_a and f_b as indicating a direction: either downward (f_a executes, then calls f_b), or upward (f_b is called, then f_a is executed). Using this we compute, for each direction, how many events in sequence can be covered by going through the edge in that direction. For instance, if $f_a \in R_{alloc}$ and $f_b \in R_{free} \cap R_{use}$, then taking the $f_a \rightarrow f_b$ call edge possibly allows to cover the three UAF events in sequence. To find double free, we also include, in this computation, call edges that allow to reach two *free* events in sequence.

Then, we favor a call edge from f_a to f_b by decreasing its weight, based on how many events in sequence the edge allows to cover. In our experiments, we use the following $\Theta_{UAF}(f_a, f_b)$ function, with a value of $\beta = 0.25$:

$$\Theta_{UAF}(f_a, f_b) \triangleq \begin{cases} \beta & \text{if } f_a \rightarrow f_b \text{ covers more than} \\ & \text{2 UAF events in sequence} \\ 1 & \text{otherwise} \end{cases}$$

Figure 5 presents an example of call graph with edges favored using the above Θ_{UAF} function.

Finally, we combine our edge weight modification with that of HAWKEYE:

$$w_{UAFuzz}(f_a, f_b) \triangleq w_{Hawkeye}(f_a, f_b) \cdot \Theta_{UAF}(f_a, f_b)$$

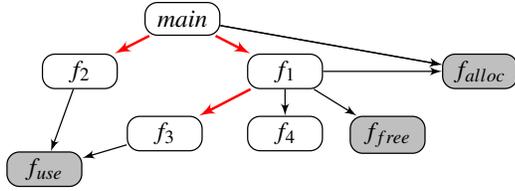


Figure 5: Example of a call graph. Favored edges are in red.

Like AFLGO, we favor the shortest path leading to the targets, since it is more likely to involve only a small number of control flow constraints, making it easier to cover by fuzzing. Our distance-based technique therefore considers both calling relations in general, via $w_{Hawkeye}$, and calling relations covering UAF events in sequence, via Θ_{UAF} .

4.4 Power Schedule

Coverage-guided fuzzers employ a power schedule (or energy assignment) to determine the number of extra inputs to generate from a selected input, which is called the *energy* of the seed. It measures how long we should spend fuzzing a particular seed. While AFL [1] mainly uses execution trace characteristics such as trace size, execution speed of the PUT and time added to the fuzzing queue for seed energy allocation, recent work [26, 48, 62] including both directed and coverage-guided fuzzing propose different power schedules. AFLGO employs simulated annealing to assign more energy for seeds closer to target locations (using the seed distance), while HAWKEYE accounts for both shorter and longer traces leading to the targets via a power schedule based on trace distance and similarity at function level.

We propose here a new power schedule using the intuitions that we should assign more energy to seeds in these cases:

- seeds that are closer (using the seed distance, Section 4.3.2);
- seeds that are more similar to the target (using the target similarity, Section 4.2.2);
- *seeds that make better decisions at critical code junctions.*

We define hereafter a new metric to evaluate the latter case.

4.4.1 Cut-edge Coverage Metric

To track progress of a seed during the fuzzing process, a fine-grained approach would consist in instrumenting the execution to compare the similarity of the execution trace of the current seed with the target bug trace, at the basic block level. But this method would slow down the fuzzing process due to high runtime overhead, especially for large programs. A more coarse-grained approach, on the other hand, is to measure the similarity at function level as proposed in HAWKEYE [28]. However, a callee can occur multiple times from different locations of single caller. Also, reaching a target function does not mean reaching the target basic blocks in this function.

Thus, we propose the lightweight *cut-edge coverage metric*, hitting a middle ground between the two aforementioned approaches by measuring progress *at the edge level* but on

the critical decision nodes only.

Algorithm 3: Accumulating cut edges

Input : Program P ; dynamic calling tree T of a bug trace

Output : Set of cut edges E_{cut}

```

1  $E_{cut} \leftarrow \emptyset$ ;
2  $nodes \leftarrow \text{flatten}(T)$ ;
3 for  $n \in nodes \wedge pn$  the node before  $n$  in  $T$  do
4   if  $n.func == pn.func$  then
5      $ce \leftarrow \text{calculate\_cut\_edges}(n.func, pn.bb, n.bb)$ ;
6   else if  $pn$  is a call to  $n.func$  then
7      $ce \leftarrow \text{calculate\_cut\_edges}(n.func, n.func.entry\_bb, n.bb)$ ;
8    $E_{cut} \leftarrow E_{cut} \cup ce$ ;
9 return  $E_{cut}$ ;

```

Algorithm 4: calculate_cut_edges inside a function

Input : A function f ; Two basic blocks bb_{source} and bb_{sink} in f

Output : Set of cut edges ce

```

1  $ce \leftarrow \emptyset$ ;
2  $cfg \leftarrow \text{get\_CFG}(f)$ ;
3  $decision\_nodes \leftarrow \{dn : \exists \text{ a path } bb_{source} \rightarrow^* dn \rightarrow^* bb_{sink} \text{ in } cfg\}$ 
4 for  $dn \in decision\_nodes$  do
5    $outgoing\_edges \leftarrow \text{get\_outgoing\_edges}(cfg, dn)$ ;
6   for  $edge \in outgoing\_edges$  do
7     if  $\text{reachable}(cfg, edge, bb_{sink})$  then
8        $ce \leftarrow ce \cup \{edge\}$ ;
9 return  $ce$ ;

```

Definition 2. A *cut edge* between two basic blocks *source* and *sink* is an outgoing edge of a decision node so that there exists a path starting from *source*, going through this edge and reaching *sink*. A *non-cut edge* is an edge which is not a cut-edge, i.e. for which there is no path from *source* to *sink* that go through this edge.

Algorithm 3 shows how cut/non-cut edges are identified in UAFUZZ given a tested binary program and an expected UAF bug trace. The main idea is to identify and accumulate the cut edges between all consecutive nodes in the (flattened) bug trace. For instance in the bug trace of Figure 3, we would first compute the cut edges between 0 and 1, then those between 1 and 2, etc. As the bug trace is a sequence of stack traces, most of the locations in the trace are “call” events, and we compute the cut edge from the function entry point to the call event in that function. However, because of the flattening, sometimes we have to compute the cut edges between different points in the same function (e.g. if in the bug trace the same function is calling *alloc* and *free*, we would have to compute the edge from the call to *alloc* to the call to *free*).

Algorithm 4 describes how cut-edges are computed inside a single function. First we have to collect the decision nodes, i.e. conditional jumps between the source and sink basic blocks. This can be achieved using a simple data-flow analysis. For each outgoing edge of the decision node, we check whether they allow to reach the sink basic block; those that can are cut edges, and the others are non-cut edges. Note that this

program analysis is intra-procedural, so that we do not need construct an inter-procedural CFG.

Our heuristic is that an input exercising more cut edges and fewer non-cut edges is more likely to cover more locations of the target. Let $E_{cut}(T)$ be the set of all cut edges of a program given the expected UAF bug trace T . We define the cut-edge score $e_s(s, T)$ of seed s as

$$e_s(s, T) \triangleq \sum_{e \in E_{cut}(T)} [(\log_2 \text{hit}(e) + 1)] - \delta * \sum_{e \notin E_{cut}(T)} [(\log_2 \text{hit}(e) + 1)]$$

where $\text{hit}(e)$ denotes the number of times an edge e is exercised, and $\delta \in (0, 1)$ is the weight penalizing seeds covering non-cut edges. In our main experiments, we use $\delta = 0.5$ according to our preliminary experiments. To deal with the path explosion induced by loops, we use bucketing [1]: the hit count is bucketized to small powers of two.

4.4.2 Energy Assignment

We propose a power schedule function that assigns energy to a seed using a combination of the three metrics that we have proposed: the prefix target similarity metric $t_p(s, T)$ (Section 4.2.2), the UAF-based seed distance $d(s, T)$ (Section 4.3.2), and the cut-edge coverage metric $e_s(s, T)$ (Section 4.4.1). The idea of our power schedule is to assign energy to a seed s proportionally to the number of targets covered in sequence $t_p(s, T)$, with a corrective factor based on seed distance d and cut-edge coverage e_s . Indeed, our power function (corresponding to `ASSIGN_ENERGY` in Algorithm 1) is defined as:

$$p(s, T) \triangleq (1 + t_p(s, T)) \times \tilde{e}_s(s, T) \times (1 - \tilde{d}_s(s, T))$$

Because their actual value is not as meaningful as the length of the covered prefix, but they allow to rank the seeds, we apply a min-max normalization [2] to get a *normalized seed distance* ($\tilde{d}_s(s, T)$) and *normalized cut-edge score* ($\tilde{e}_s(s, T)$). For example, $\tilde{d}_s(s, T) = \frac{d_s(s, T) - \min D}{\max D - \min D}$ where $\min D$, $\max D$ denote the minimum and maximum value of seed distance so far. Note that both metric scores are in $(0, 1)$, i.e. can only reduce the assigned energy when their score is bad.

4.5 Postprocess and Bug Triage

Since UAF bugs are often silent, all seeds generated by a directed fuzzer must *a priori* be sent to a *bug triager* (typically, a profiling tool such as VALGRIND) in order to confirm whether they are bug triggering input or not. Yet, this can be extremely expensive as fuzzers generate a huge amount of seeds and bug triagers are expensive.

Fortunately, the target similarity metric allows UAFUZZ to compute the sequence of covered targets of each fuzzed input at runtime. This information is *available for free* for each seed once it has been created and executed. We capitalize on it in order to *pre-identify* likely-bug triggering seeds, i.e. seeds that indeed cover the three UAF events in sequence. Then, the bug triager is run only over these pre-identified seeds, the other ones being simply discarded – potentially saving a huge

amount of time in bug triaging.

5 Implementation

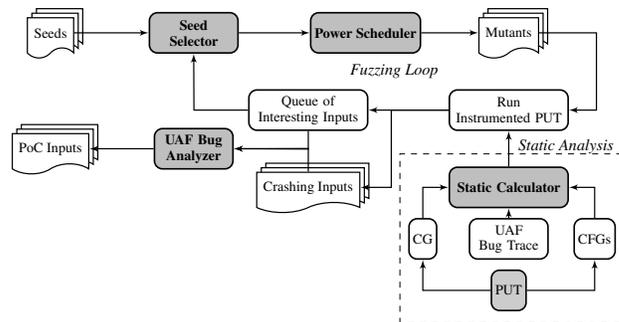


Figure 6: Overview of UAFUZZ workflow.

We implement our results in a UAF-oriented binary-level directed fuzzer named UAFUZZ. Figure 6 depicts an overview of the main components of UAFUZZ. The input of the overall system are a set of initial seeds, the PUT in binary and target locations extracted from the bug trace. The output is a set of unique bug-triggering inputs. The prototype is built upon AFL 2.52b [1] and QEMU 2.10.0 for fuzzing, and the binary analysis platform BINSEC [4] for (lightweight) static analysis. These two components share information such as target locations, time budget and fuzzing status.

6 Experimental Evaluation

6.1 Research Questions

To evaluate the effectiveness and efficiency of our approach, we investigate four principal research questions:

RQ1. UAF Bug-reproducing Ability Can UAFUZZ outperform other directed fuzzing techniques in terms of UAF bug reproduction in executables?

RQ2. UAF Overhead How does UAFUZZ compare to other directed fuzzing approaches w.r.t. instrumentation time and runtime overheads?

RQ3. UAF Triage How much does UAFUZZ reduce the number of inputs to be sent to the bug triage step?

RQ4. Individual Contribution How much does each UAFUZZ component contribute to the overall results?

We will also evaluate UAFUZZ in the context of *patch testing*, another important application of directed fuzzing [25, 28, 59].

6.2 Evaluation Setup

Evaluation Fuzzers. We aim to compare UAFUZZ with state-of-the-art directed fuzzers, namely AFLGO [2] and HAWKEYE [28], using AFL-QEMU as a baseline (binary-level coverage-based fuzzing). Unfortunately, both AFLGO and HAWKEYE work on source code, and while AFLGO is open source, HAWKEYE is not available. Hence, we *implemented binary-level versions* of AFLGO and HAWKEYE, coined as

AFLGOB and HAWKEYEB. We closely follow the original papers, and, for AFLGO, use the source code as a reference. AFLGOB and HAWKEYEB are implemented on top of AFL-QEMU, following the generic architecture of UAFUZZ but with dedicated distance, seed selection and power schedule mechanisms. Table 2 summarizes our different fuzzer implementations and a comparison with their original counterparts.

Table 2: Overview of main techniques of greybox fuzzers. Our own implementations are marked with *.

Fuzzer	Directed	Binary?	Distance	Seed Selection	Power Schedule	Mutation
AFL-QEMU	✗	✓	–	AFL	AFL	AFL
AFLGO	✓	✗	CFG-based	~ AFL	Annealing	~ AFL
AFLGOB*	✓	✓	~ AFLGO	~ AFLGO	~ AFLGO	~ AFLGO
HAWKEYE	✓	✗	AADF	distance-based	Trace fairness	Adaptive
HAWKEYEB*	✓	✓	~ HAWKEYE	~ HAWKEYE	≈ HAWKEYE	~ AFLGO
UAFUZZ*	✓	✓	UAF-based	Targets-based	UAF-based	~ AFLGO

We evaluate the implementation of AFLGOB (Appendix B, Appendix) and find it very close to the original AFLGO after accounting for emulation overhead.

UAF Fuzzing Benchmark. The standard UAF micro benchmark Juliet Test Suite [56] for static analyzers is too simple for fuzzing. No macro benchmark actually assesses the effectiveness of UAF detectors – the widely used LAVA [36] only contains buffer overflows. Thus, we construct a new UAF benchmark according to the following rationale:

1. The subjects are real-world popular and fairly large security-critical programs;
2. The benchmark includes UAF bugs found by existing fuzzers from the fuzzing literature [1, 26, 28, 40] or collected from NVD [20]. Especially, we include *all* UAF bugs found by directed fuzzers;
3. The bug report provides detailed information (e.g., buggy version and the stack trace), so that we can identify target locations for fuzzers.

In summary, we have 13 known UAF vulnerabilities (2 from directed fuzzers) over 11 real-world C programs whose sizes vary from 26 Kb to 3.8 Mb. Furthermore, selected programs range from image processing to data archiving, video processing and web development. Our benchmark is therefore representative of different UAF vulnerabilities of real-world programs. Table 3 presents our evaluation benchmark.

Evaluation Configurations. We follow the recommendations for fuzzing evaluations [43] and use the same fuzzing configurations and hardware resources for all experiments. Experiments are conducted 10 times with a time budget depending on the PUT. We use as input seed either an empty file or existing valid files provided by developers. We do not use any token dictionary. All experiments were carried out on an Intel Xeon CPU E3-1505M v6 @ 3.00GHz CPU with 32GB RAM and Ubuntu 16.04 64-bit.

6.3 UAF Bug-reproducing Ability (RQ1)

Protocol. We compare the different fuzzers on our 13 UAF vulnerabilities using *Time-to-Exposure* (TTE), i.e. the time

Table 3: Overview of our evaluation benchmark

Bug ID	Program		Bug		#Targets in trace
	Project	Size	Type	Crash	
giflib-bug-74	GIFLIB	59 Kb	DF	✗	7
CVE-2018-11496	lrzip	581 Kb	UAF	✗	12
yasm-issue-91	yasm	1.4 Mb	UAF	✗	19
CVE-2016-4487	Binutils	3.8 Mb	UAF	✓	7
CVE-2018-11416	jpegtoptim	62 Kb	DF	✗	5
mjs-issue-78	mjs	255 Kb	UAF	✗	19
mjs-issue-73	mjs	254 Kb	UAF	✗	28
CVE-2018-10685	lrzip	576 Kb	UAF	✗	7
CVE-2019-6455	Recutils	604 Kb	DF	✗	15
CVE-2017-10686	NASM	1.8 Mb	UAF	✓	10
gifsicle-issue-122	Gifsicle	374 Kb	DF	✗	11
CVE-2016-3189	bzip2	26 Kb	UAF	✓	5
CVE-2016-20623	Binutils	1.0 Mb	UAF	✗	7

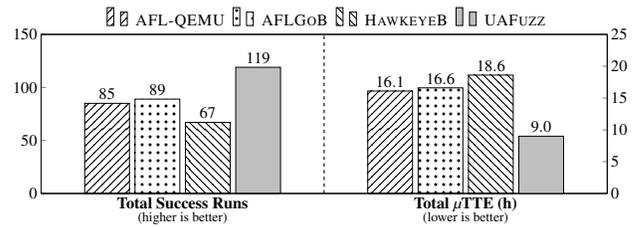


Figure 7: Summary of fuzzing performance (RQ1)

elapsed until first bug-triggering input, and *number of success runs* in which a fuzzer triggers the bug. In case a fuzzer cannot detect the bug within the time budget, the run’s TTE is set to the time budget. Following existing work [25, 28], we use the *Vargha-Delaney statistic* (\hat{A}_{12}) metric [69]⁶ to assess the confidence that one tool outperforms another. Code coverage is not relevant for directed fuzzers.

Results. Figure 7 presents a consolidated view of the results (total success runs and TTE – we denote by μ TTE the average TTE observed for each sample over 10 runs). Appendix A contains additional information about consolidated Vargha-Delaney statistics (Table 4).

Figure 7 (and Table 4) show that UAFUZZ clearly outperforms the other fuzzers both in total success runs (vs. 2nd best AFLGOB: +34% in total, up to +300%) and in TTE (vs. 2nd best AFLGOB, total: 2.0 \times , avg: 6.7 \times , max: 43 \times). In some specific cases, UAFUZZ saves roughly 10,000s of TTE over AFLGOB or goes from 0/10 successes to 7/10. The \hat{A}_{12} value of UAFUZZ against other fuzzers is also significantly above the conventional large effect size 0.71 [69], as shown in Table 4 (vs. 2nd best AFLGOB, avg: 0.78, median: 0.80, min: 0.52).

UAFUZZ *significantly* outperforms state-of-the-art directed fuzzers in terms of UAF bugs reproduction with a high confidence level.

Note that performance of AFLGOB and HAWKEYEB w.r.t. their original source-level counterparts are representative (cf. Appendix B).

⁶Value between 0 and 1, the higher the better. Values above the conventionally large effect size of 0.71 are considered highly relevant [69].

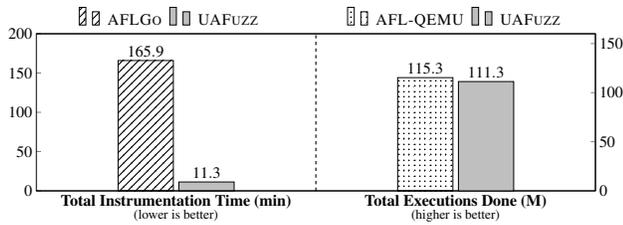


Figure 8: Global overhead (RQ2)

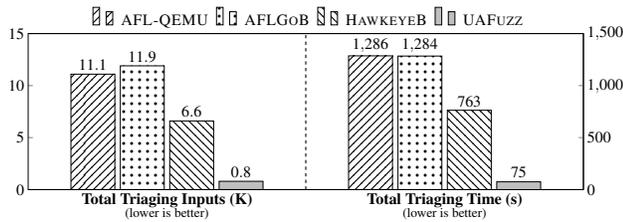


Figure 9: Summary of bugs triage (RQ3)

6.4 UAF Overhead (RQ2)

Protocol. We are interested in both (1) instrumentation-time overhead and (2) runtime overhead. For (1), we simply compute the total instrumentation time of UAFUZZ and we compare it to the instrumentation time of AFLGO. For (2), we compute the total number of executions per second of UAFUZZ and compare it to AFL-QEMU taken as a baseline.

Results. Consolidated results for both instrumentation-time and runtime overhead are presented in Figure 8 (number of executions per second is replaced by the total number of executions performed in the same time budget). This figure shows that UAFUZZ is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer AFLGO in the instrumentation phase*, and has almost the same total number of executions per second as AFL-QEMU. Appendix C contains additional results with detailed instrumentation time (Figure 12) and runtime statistics (Figure 14), as well as instrumentation time for AFLGoB and HAWKEYEB (Figure 13).

UAFUZZ enjoys both a *lightweight instrumentation time* and a *minimal runtime overhead*.

6.5 UAF Triage (RQ3)

Protocol. We consider the total number of triaging inputs (number of inputs sent to the triaging step), the triaging inputs rate TIR (ratio between the total number of generated inputs and those sent to triaging) and the total triaging time (time spent within the triaging step). Since other fuzzers cannot identify inputs reaching targets during the fuzzing process, we conservatively analyze *all* inputs generated by these fuzzers in the bug triage step (TIR = 1).

Results. Consolidated results are presented in Figure 9, detailed results in Appendix D, Table 6 and Figure 15.

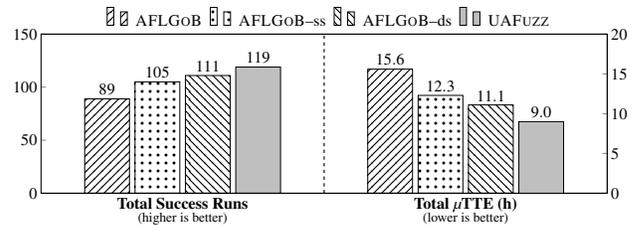


Figure 10: Impact of each component (RQ4)

- The TIR of UAFUZZ is 9.2% in total (avg: 7.25%, median: 3.14%, best: 0.24%, worst: 30.22%) – sparing up to 99.76% of input seeds for confirmation, and is always less than 9% except for sample `mjs`;
- Figure 15 shows that UAFUZZ spends the smallest amount of time in bug triage, i.e. 75s (avg: 6s, min: 1s, max: 24s) for a total speedup of 17 \times over AFLGoB (max: 130 \times , avg: 39 \times).

UAFUZZ reduces a *large* portion (i.e., more than 90%) of triaging inputs in the post-processing phase. Subsequently, UAFUZZ only spends several seconds in this step, winning an order of magnitude compared to standard directed fuzzers.

6.6 Individual Contribution (RQ4)

Protocol. We compare four different versions of our prototype, representing a continuum between AFLGO and UAFUZZ: (1) the basic AFLGO represented by AFLGoB, (2) AFLGoB-ss adds our seed selection metric to AFLGoB, (3) AFLGoB-ds adds the UAF-based function distance to AFLGoB-ss, and finally (4) UAFUZZ adds our dedicated power schedule to AFLGoB-ds. We consider the previous RQ1 metrics: number of success runs, TTE and Vargha-Delaney. Our goal is to assess whether or not these technical improvements do lead to fuzzing performance improvements.

Results. Consolidated results for success runs and TTE are represented in Figure 10. Appendix E includes detailed results plus Vargha-Delaney metric (Table 7).

As summarized in Figure 10, we can observe that each new component does improve both TTE and number of success runs, leading indeed to fuzzing improvement. Detailed results in Table 7 with \hat{A}_{12} values show the same clear trend.

The UAF-based distance computation, the power scheduling and the seed selection heuristic *individually* contribute to improve fuzzing performance, and combining them yield even further improvements, demonstrating their interest and complementarity.

6.7 Patch Testing & Zero-days

Patch testing. The idea is to use bug stack traces of *known* UAF bugs to guide testing on the *patched* version of the PUT

– instead of the buggy version as in bug reproduction. The benefit from the bug hunting point of view [17] is both to try finding buggy or incomplete patches *and* to focus testing on *a priori* fragile parts of the code, possibly discovering bugs unrelated to the patch itself.

How to. We follow bug hunting practice [17]. Starting from the recent publicly disclosed UAF bugs of open source programs, we manually identify addresses of relevant call instructions in the reported bug stack traces since the code has been evolved. We focus mainly on 3 widely-used programs that have been well fuzzed and maintained by the developers, namely GNU patch, GPAC and Perl 5 (737K lines of C code and 5 known bug traces in total). We also consider 3 other codes: MuPDF, Boolector and fontforge (+1,196Kloc).

Results. Overall UAFUZZ has found and reported **30 new bugs**, including **11 new UAF bugs** and **7 new CVE** (details in Appendix F, Table 8). *At this time, 17 bugs have been fixed by the vendors.* Interestingly, the bugs found in GNU patch (Appendix F) and GPAC were actually *buggy patches*.

UAFUZZ has been proven effective in a patch testing setting, allowing to find 30 new bugs (incl. 7 new CVE) in 6 widely-used programs.

6.8 Threats to Validity

Implementation. Our prototype is implemented as part of the binary-level code analysis framework BINSEC [34, 35], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [22,33,63], and on top of the popular fuzzer AFL-QEMU. Effectiveness and correctness of UAFUZZ have been assessed on several bug traces from real programs, as well as on small samples from the Juliet Test Suite. All reported UAF bugs have been manually checked.

Benchmark. Our benchmark is built on both real codes *and* real bugs, and encompass several bugs found by recent fuzzing techniques of well-known open source codes (including all UAF bugs found by directed fuzzers).

Competitors. We consider the best state-of-the-art techniques in directed fuzzing, namely AFLGO [25] and HAWKEYE [28]. Unfortunately, HAWKEYE is not available and AFLGO works on source code only. Thus, we re-implement these technologies in our own framework. We followed the available information (article, source code if any) as close as possible, and did our best to get precise implementations. They have both been checked on real programs and small samples, and the comparison against AFLGO source (Appendix B) and our own AFLGOB implementation is conclusive.

7 Related Work

Directed Greybox Fuzzing. AFLGO [25] and HAWKEYE [28] have already been discussed. LOLLY [49] provides a lightweight instrumentation to measure the sequence

basic block coverage of inputs, yet, at the price of a large runtime overhead. SEEDEDFUZZ [71] seeks to generate a set of initial seeds that improves directed fuzzing performance. SEMFUZZ [74] leverages vulnerability-related texts such as CVE reports to guide fuzzing. 1DVUL [59] discovers 1-day vulnerabilities via binary patches.

UAFUZZ is the first directed fuzzer tailored to UAF bugs, and one of the very few [59] able to handle binary code.

Coverage-based Greybox Fuzzing. AFL [1] is the seminal coverage-guided greybox fuzzer. Substantial efforts have been conducted in the last few years to improve over it [26,40,46]. Also, many efforts have been fruitfully invested in combining fuzzing with other approaches, such as static analysis [40,47], dynamic taint analysis [29,30,62], symbolic execution [58,67,76] or machine learning [41,66].

Recently, UAFL [70] - another independent research effort on the same problem, specialized coverage-guided fuzzing to detect UAFs by finding operation sequences potentially violating a tpestate property and then guiding the fuzzing process to trigger property violations. However, this approach relies heavily on the static analysis of source code, therefore is not applicable at binary-level.

Our technique is orthogonal to all these improvements, they could be reused within UAFUZZ as is.

UAF Detection. Precise static UAF detection is difficult. GUEB [12] is the only binary-level static analyzer for UAF. The technique can be combined with dynamic symbolic execution to generate PoC inputs [38], yet with scalability issues. On the other hand, several UAF source-level static detectors exist, based on abstract interpretation [32], pointer analysis [72], pattern matching [57], model checking [44] or demand-driven pointer analysis [68]. A common weakness of all static detectors is their inability to infer triggering input – they rather prove their absence.

Dynamic UAF detectors mainly rely on heavyweight instrumentation [9,27,55] and result in high runtime overhead, even more for closed source programs. ASan [65] performs lightweight instrumentation, but at source level only.

UAF Fuzzing Benchmark. While the Juliet Test Suite [56] (CWE-415, CWE-416)⁷ contains only too small programs, popular fuzzing benchmarks [7,11,16,36,64] comprise only very few UAF bugs. Moreover, many of these benchmarks contain either artificial bugs [7,16,36,64] or artificial programs [56].

Merging our evaluation benchmark (known UAF) and our new UAF bugs, we provide the largest fuzzing benchmark dedicated to UAF – 17 real codes and 30 real bugs

8 Conclusion

UAFUZZ is the *first directed* greybox fuzzing approach tailored to detecting UAF vulnerabilities (in binary) given only the bug stack trace. UAFUZZ outperforms existing directed fuzzers, both in terms of time to bug exposure and number of

⁷Juliet is mostly used for the evaluation of C/C++ static analysis tools.

successful runs. UAFUZZ has been proven effective in both bug reproduction and patch testing. We release the source code of UAFUZZ and the UAF fuzzing benchmark at:

<https://github.com/strongcourage/uafuzz>
<https://github.com/strongcourage/uafbench>

Acknowledgement

This work was supported by the H2020 project C4IoT under the Grant Agreement No 833828 and FUI CAESAR.

References

- [1] Afl. <http://lcamtuf.coredump.cx/afl/>, 2020.
- [2] Aflgo. <https://github.com/aflgo/aflgo>, 2020.
- [3] Aflgo's issues. <https://github.com/aflgo/aflgo/issues>, 2020.
- [4] Binsec. <https://binsec.github.io/>, 2020.
- [5] Circumventing fuzzing roadblocks with compiler transformations. <https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/>, 2020.
- [6] Cve-2018-6952. <https://savannah.gnu.org/bugs/index.php?53133>, 2020.
- [7] Darpa cgc corpus. <http://www.lungetech.com/2017/04/24/cgc-corpus/>, 2020.
- [8] Double free in gnu patch. <https://savannah.gnu.org/bugs/index.php?56683>, 2020.
- [9] Dr.memory. <https://www.drmemory.org/>, 2020.
- [10] Gnu patch. <https://savannah.gnu.org/projects/patch/>, 2020.
- [11] Google fuzzer testsuite. <https://github.com/google/fuzzer-test-suite>, 2020.
- [12] Gueb: Static analyzer detecting use-after-free on binary. <https://github.com/montyly/gueb>, 2020.
- [13] Libfuzzer. <https://llvm.org/docs/LibFuzzer.html>, 2020.
- [14] OSS-Fuzz: Continuous Fuzzing Framework for Open-Source Projects. <https://github.com/google/oss-fuzz/>, 2020.
- [15] Oss-fuzz: Five months later, and rewarding projects. <https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html>, 2020.
- [16] Rode0day. <https://rode0day.mit.edu/>, 2020.
- [17] Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. <https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html>, 2020.
- [18] Uaf fuzzing benchmark. <https://github.com/strongcourage/uafbench>, 2020.
- [19] Uafuzz. <https://github.com/strongcourage/uafuzz>, 2020.
- [20] Us national vulnerability database. <https://nvd.nist.gov/vuln/search>, 2020.
- [21] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.
- [22] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. 2017.
- [23] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. Grimoire: synthesizing structure while fuzzing. In *USENIX Security Symposium (USENIX Security 19)*, 2019.
- [24] Marcel Böhme. Assurance in software testing: A roadmap. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '19*, 2019.
- [25] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.
- [26] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [27] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.
- [28] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [29] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.
- [30] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. *arXiv preprint arXiv:1905.12228*, 2019.
- [31] Maria Christakis, Peter Müller, and Valentin Wüstholtz. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.
- [32] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, 2012.
- [33] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA*, 2016.
- [34] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, 2016.
- [35] Adel Djoudi and Sébastien Bardin. Binsec: Binary code analysis with low-level regions. In *TACAS*, 2015.
- [36] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, 2016.
- [37] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bunt. The rode0day to less-buggy programs. *IEEE Security & Privacy*, 2019.
- [38] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, 2016.
- [39] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, 2020.
- [40] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.
- [41] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

- [42] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, 2012.
- [43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [44] Daniel Kroening and Michael Tautschnig. Cbmc-c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014.
- [45] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.
- [46] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.
- [47] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.
- [48] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
- [49] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension*, 2019.
- [50] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
- [51] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.
- [52] Microsoft. Project springfield. <https://www.microsoft.com/en-us/security-risk-detection/>, 2020.
- [53] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 1990.
- [54] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan*, 2007.
- [56] NIST. Juliet test suite for c/c++. <https://samate.nist.gov/SARD/testsuite.php>, 2020.
- [57] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. Coccinelle: tool support for automated cert c secure coding standard certification. *Science of Computer Programming*, 2014.
- [58] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.
- [59] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.
- [60] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.
- [61] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [62] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
- [63] Frédéric Recoules, Sébastien Bardin, Richard Bonichon Bonichon, Laurent Mounier, and Marie-Laure Potet. Get rid of inline assembly through verification-oriented lifting. In *ASE*, 2019.
- [64] Subhajt Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.
- [65] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.
- [66] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.
- [67] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.
- [68] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.
- [69] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 2000.
- [70] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *42nd International Conference on Software Engineering*, 2020.
- [71] Weiguang Wang, Hao Sun, and Qingkai Zeng. Seededfuzz: Selecting and generating seeds for directed fuzzing. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2016.
- [72] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.
- [73] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (SP)*, 2019.
- [74] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.
- [75] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.
- [76] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

A UAF Bug-reproducing Ability (RQ1)

We present in this section additional results regarding RQ1 including more detailed experimental reports.

Experimental results. Table 4 summarizes the fuzzing performance of 4 binary-based fuzzers against our benchmark by providing the total number of covered paths, the total number of success runs and the max/min/average/median values of *Factor* and \hat{A}_{12} . Table 5 compares our fuzzer UAFUZZ with several variants of directed fuzzers AFLGO.

Table 4: Summary of bug reproduction of UAFUZZ compared to other fuzzers against our fuzzing benchmark. Statistically significant results $\hat{A}_{12} \geq 0.71$ are marked as bold.

Fuzzer	Total Avg Paths	Success Runs	Factor				\hat{A}_{12}			
			Mdn	Avg	Min	Max	Mdn	Avg	Min	Max
AFL-QEMU	10.6K	85 (+40%)	2.01	6.66	0.60	46.63	0.82	0.78	0.29	1.00
AFLGOB	11.1K	89 (+34%)	1.96	6.73	0.96	43.34	0.80	0.78	0.52	1.00
HAWKEYEB	7.3K	67 (+78%)	2.90	8.96	1.21	64.29	0.88	0.86	0.56	1.00
UAFUZZ	8.2K	119	--	--	--	--	--	--	--	--

Table 5: Bug reproduction of AFLGO against our benchmark except CVE-2017-10686 due to compilation issues of AFLGO. Numbers in red are the best μ TTEs.

Bug ID	AFLGO (source)		AFLGO _F (source)		AFLGO _B		UAFUZZ	
	Runs	μ TTE(s)	Runs	μ TTE(s)	Runs	μ TTE(s)	Runs	μ TTE(s)
giflib-bug-74	10	62	10	281	9	478	10	209
CVE-2018-11496	10	2	10	38	10	22	10	14
yasm-issue-91	10	307	8	2935	8	2427	10	56
CVE-2016-4487	10	676	10	1386	6	2427	6	2110
CVE-2018-11416	10	78	7	1219	10	303	10	235
mjs-issue-78	10	1417	3	9706	4	8755	9	4197
mjs-issue-73	9	5207	3	34210	0	10800	7	4881
CVE-2018-10685	10	74	9	1072	9	305	10	156
CVE-2019-6455	5	1090	0	20296	5	1213	10	438
gifsicle-issue-122	8	4161	7	25881	6	9811	7	9853
CVE-2016-3189	10	72	10	206	10	158	10	141
CVE-2018-20623	10	177	10	1329	9	3169	10	128
Total Success Runs	112		87		86		109	
Total μTTE (h)	3.7		27.4		10.1		6.2	

B Regarding implementations of AFLGO_B and HAWKEYEB

Comparison between AFLGO_B and source-based AFLGO. We want to evaluate how close our implementation of AFLGO_B is from the original AFLGO, in order to assess the degree of confidence we can have in our results – we do not do it for HAWKEYEB as HAWKEYE is not available.

AFLGO unsurprisingly performs better than AFLGO_B and UAFUZZ (Figure 11, Table 5 in Appendix). This is largely due to the emulation runtime overhead of QEMU, a well-documented fact. Still, *surprisingly enough*, UAFUZZ can find the bugs faster than AFLGO in 4 samples, demonstrating its efficiency.

Yet, more interestingly, Figure 11 also shows that once emulation overhead⁸ is taken into account (yielding AFLGO_F, the expected *binary-level* performance of AFLGO), then AFLGO_B is in line with AFLGO_F (and even shows better TTE) – UAFUZZ even significantly outperforms AFLGO_F.

⁸We estimate for each sample an overhead factor f by comparing the number of executions per second in both AFL and AFL-QEMU, then multiply the computation time of AFLGO by $f - f$ varies from 2.05 to 22.5 in our samples.

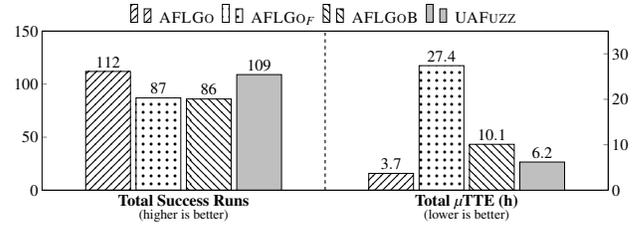


Figure 11: Summary of fuzzing performance of 4 fuzzers against our benchmark, except CVE-2017-10686 due to compilation issues of AFLGO.

Performance of AFLGO_B is in line with the original AFLGO once QEMU overhead is taken into account, allowing a fair comparison with UAFUZZ. UAFUZZ nonetheless performs relatively well on UAF compared with the source-based directed fuzzer AFLGO, demonstrating the benefit of our original fuzzing mechanisms.

About performance of HAWKEYEB in RQ1. HAWKEYEB performs significantly worse than AFLGO_B and UAFUZZ in §6.3. We cannot compare HAWKEYEB with HAWKEYE as HAWKEYE is not available. Still, we investigate that issue and found that this is mostly due to a large runtime overhead spent calculating the target similarity metric. Indeed, according to the HAWKEYE original paper [28], this computation involves some *quadratic computation* over the *total number of functions* in the code under test. On our samples this number quickly becomes important (up to 772) while the number of targets (UAFUZZ) remains small (up to 28). A few examples: CVE-2017-10686: 772 functions vs 10 targets; gifsicle-issue-122: 516 functions vs 11 targets; mjs-issue-78: 450 functions vs 19 targets. Hence, we can conclude that on our samples the performance of HAWKEYEB are in line with what is expected from HAWKEYE algorithm.

C UAF Overhead (RQ2)

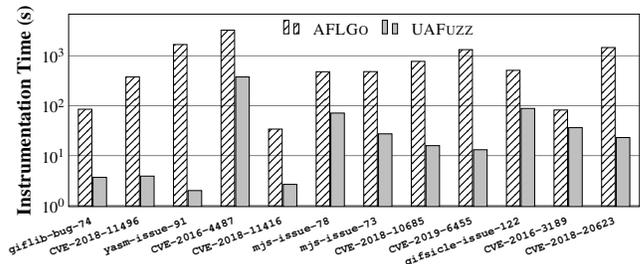


Figure 12: Average instrumentation time in seconds (except CVE-2017-10686 due to compilation issues of AFLGO).

Additional data. We first provide additional results for RQ2. Figures 12 and 13 compare the average instrumentation time between, respectively, UAFUZZ and the source-based directed

fuzzer AFLGO; and UAFUZZ and the two binary-based directed fuzzers AFLGOB and HAWKEYEB. Figure 14 shows the total execution done of AFL-QEMU and UAFUZZ for each subject in our benchmark.

Detailed results. We now discuss experimental results regarding overhead in more depth than what was done in §6.4.

- Figures 8 and 12 show that UAFUZZ is *an order of magnitude faster than the state-of-the-art source-based directed fuzzer AFLGO in the instrumentation phase* (14.7× faster in total). For example, UAFUZZ spends only 23s (i.e., 64× less than AFLGO) in processing the large program `readelf` of Binutils;
- Figures 8 and 14 show that UAFUZZ has almost the same total number of executions per second as AFL-QEMU (-4% in total, -12% in average), meaning that its overhead is negligible.
- Figure 13 shows that HAWKEYEB is sometimes significantly slower than UAFUZZ (2×). This is mainly because of the cost of target function trace closure calculation on large examples with many functions (cf. §6.3).

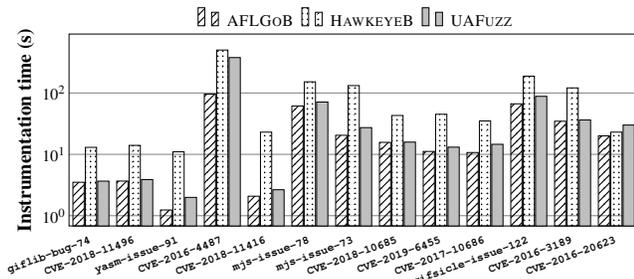


Figure 13: Average instrumentation time in seconds.

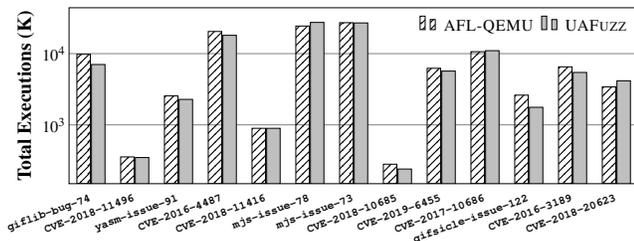


Figure 14: Total executions done in all runs.

D UAF Triage (RQ3)

We provide additional results for RQ3: Figure 15 and Table 6 show the average triaging time and number of triaging inputs (including TIR values for UAFUZZ) of 4 fuzzers against our benchmark.

E Individual Contribution (RQ4)

We provide additional results for RQ4. Table 7 shows the fuzzing performance of 2 AFLGOB-based variants

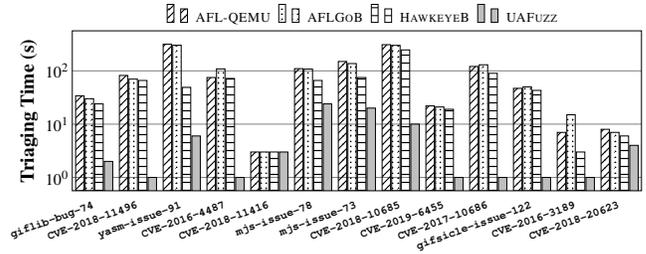


Figure 15: Average triaging time in seconds.

Table 6: Average number of triaging inputs of 4 fuzzers against our tested subjects. For UAFUZZ, the TIR values are in parentheses.

Bug ID	AFL-QEMU	AFLGOB	HAWKEYEB	UAFUZZ
gflib-bug-74	200.9	177.0	139.9	10.0 (5.31%)
CVE-2018-11496	409.6	351.7	332.5	5.4 (4.08%)
yasm-issue-91	2115.3	2023.0	326.6	37.4 (2.72%)
CVE-2016-4487	933.1	1367.2	900.2	2.5 (0.24%)
CVE-2018-11416	21.5	21.0	21.0	1.0 (4.76%)
mjs-issue-78	1226.9	1537.8	734.6	262.3 (30.22%)
mjs-issue-73	1505.6	1375.9	745.6	252.2 (29.25%)
CVE-2018-10685	414.2	402.1	328.9	12.6 (3.14%)
CVE-2019-6455	243.2	238.1	211.1	6.9 (1.57%)
CVE-2017-10686	2416.9	2517.0	1765.2	214.3 (8.96%)
gifsicle-issue-122	405.0	431.7	378.5	3.3 (0.86%)
CVE-2016-3189	377.9	764.7	126.4	7.1 (1.69%)
CVE-2018-20623	804.0	724.2	625.1	5.4 (1.39%)
Total	11.1K	11.9K	6.6K	820 (7.25%)

Table 7: Bug reproduction on 4 fuzzers against our benchmark. \hat{A}_{12A} and \hat{A}_{12U} denote the Vargha-Delaney values of AFLGOB and UAFUZZ. Statistically significant results for \hat{A}_{12} (e.g., $\hat{A}_{12A} \leq 0.29$ or $\hat{A}_{12U} \geq 0.71$) are in bold.

Fuzzers	AFLGOB	AFLGOB-ss	AFLGOB-ds	UAFUZZ
Total Success Runs	89	105 (+18.0%)	111 (+24.7%)	119 (+33.7%)
Total μ TTE (h)	15.6	12.3	11.1	9.0
Average \hat{A}_{12A}	-	0.29	0.37	0.22
Average \hat{A}_{12U}	0.78	0.54	0.64	-

AFLGOB-ss and AFLGOB-ds compared to AFLGOB and our tool UAFUZZ against our benchmark.

F Patch Testing & Zero-days

We provide additional results for patch testing (Table 8), as well as a **detailed discussion on the GNU Patch buggy patch**.

Zoom: GNU Patch buggy patch. We use CVE-2018-6952 [6] to demonstrate the effectiveness of UAFUZZ in exposing unknown UAF vulnerabilities. GNU patch [10] takes a patch file containing a list of differences and applies them to the original file. Listing 3 shows the code fragment of CVE-2018-6952 which is a double free in the latest version 2.7.6 of GNU patch. Interestingly, by using the stack trace of this CVE as shown in Figure 16, UAFUZZ successfully discovered an *incomplete bug fix* [8] CVE-2019-20633 in the latest commit 76e7758, with a slight difference of the bug stack trace (i.e., the call of `savebuf()`)

```

1 File: src/patch.c
2 int main (int argc, char **argv) {...
3 while (0 < (got_hunk = another_hunk (diff_type, reverse
4 ...)) { /* Apply each hunk of patch */ ... }
5
6 File: src/pch.c
7 int another_hunk (enum diff difftype, bool rev) { ...
8 while (p_end >= 0) {
9 if (p_end == p_efake) p_end = p_bfake;
10 else free(p_line[p_end]); /* Free and Use event */
11 p_end--;
12 } ...
13 while (p_end < p_max) { ...
14 switch(*buf) { ...
15 case '+': case '!': /* Our bug CVE-2019-20633 */ ...
16 p_line[p_end] = savebuf (s, chars_read); ...
17 case ' ': /* CVE-2018-6952 */ ...
18 p_line[p_end] = savebuf (s, chars_read); ...
19 ...}
20 ...}
21 ... }
22
23 File: src/util.c
24 /* Allocate a unique area for a string. */
25 char *savebuf (char const *s, size_t size) { ...
26 rv = malloc (size); /* Alloc event */ ...
27 memcpy (rv, s, size);
28 return rv;
29 }

```

Listing 3: Code fragment of GNU patch pertaining to the UAF vulnerability CVE-2018-6952.

in another_hunk()).

Technically, GNU patch takes an input patch file containing multiple hunks (line 3) that are split into multiple strings us-

```

==330== Invalid free() / delete / delete[] / realloc()
==330== at 0x402D358: free (in vgppreload_memcheck-x86-linux.so)
==330== by 0x8052E11: another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)
==330== Address 0x4283540 is 0 bytes inside a block of size 2 free'd
==330== at 0x402D358: free (in vgppreload_memcheck-x86-linux.so)
==330== by 0x8052E11: another_hunk (pch.c:1185)
==330== by 0x804C06C: main (patch.c:396)
==330== Block was alloc'd at
==330== at 0x402C17C: malloc (in vgppreload_memcheck-x86-linux.so)
==330== by 0x805A821: savebuf (util.c:861)
==330== by 0x805423C: another_hunk (pch.c:1504)
==330== by 0x804C06C: main (patch.c:396)

```

Figure 16: The bug trace of CVE-2018-6952 produced by VALGRIND.

ing special characters as delimiter via *buf in the switch case (line 14). GNU patch then reads and parses each string stored in p_line that is dynamically allocated on the memory using malloc() in savebuf() (line 26) until the last line of this hunk has been processed. Otherwise, GNU patch deallocates the most recently processed string using free() (line 10). Our reported bug and CVE-2018-6952 share the same free and use event, but have a different stack trace leading to the same alloc event. Actually, while the PoC input generated by UAFUZZ contains two characters '!', the PoC of CVE-2018-6952 does not contain this character, consequently the case in line 16 was previously uncovered, and thus this CVE had been incompletely fixed. This case study shows the importance of producing different unique bug-triggering inputs to favor the repair process and help complete bug fixing.

Table 8: Summary of zero-day vulnerabilities reported by our fuzzer.

Program	Code Size	Version (Commit)	Bug ID	Vulnerability Type	Crash	Vulnerable Function	Status	CVE
GPAC	545K	0.7.1 (987169b)	#1269	User after free	✗	gf_m2ts_process_pmt	Fixed	CVE-2019-20628
		0.8.0 (56eaea8)	#1440-1	User after free	✗	gf_isom_box_del	Fixed	CVE-2020-11558
		0.8.0 (56eaea8)	#1440-2	User after free	✗	gf_isom_box_del	Fixed	Pending
		0.8.0 (56eaea8)	#1440-3	User after free	✗	gf_isom_box_del	Fixed	Pending
		0.8.0 (5b37b21)	#1427	User after free	✓	gf_m2ts_process_pmt		
		0.7.1 (987169b)	#1263	NULL pointer dereference	✓	ilst_item_Read	Fixed	
		0.7.1 (987169b)	#1264	Heap buffer overflow	✓	gf_m2ts_process_pmt	Fixed	CVE-2019-20629
		0.7.1 (987169b)	#1265	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1266	Invalid read	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1267	NULL pointer dereference	✓	gf_m2ts_process_pmt	Fixed	
		0.7.1 (987169b)	#1268	Heap buffer overflow	✓	BS_ReadByte	Fixed	CVE-2019-20630
		0.7.1 (987169b)	#1270	Invalid read	✓	gf_list_count	Fixed	CVE-2019-20631
		0.7.1 (987169b)	#1271	Invalid read	✓	gf_odf_delete_descriptor	Fixed	CVE-2019-20632
		0.8.0 (5b37b21)	#1445	Heap buffer overflow	✓	gf_bs_read_data	Fixed	
0.8.0 (5b37b21)	#1446	Stack buffer overflow	✓	gf_m2ts_get_adaptation_field	Fixed			
GNU patch	7K	2.7.6 (76e7758)	#56683	Double free	✓	another_hunk	Confirmed	CVE-2019-20633
		2.7.6 (76e7758)	#56681	Assertion failure	✓	pch_swap	Confirmed	
		2.7.6 (76e7758)	#56684	Memory leak	✗	xmalloc	Confirmed	
Perl 5	184K	5.31.3 (a3c7756)	#134324	User after free	✓	S_reg	Confirmed	
		5.31.3 (a3c7756)	#134326	User after free	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134329	User after free	✓	Perl_regnext	Fixed	
		5.31.3 (a3c7756)	#134322	NULL pointer dereference	✓	do_clean_named_objs	Confirmed	
		5.31.3 (a3c7756)	#134325	Heap buffer overflow	✓	S_reg	Fixed	
		5.31.3 (a3c7756)	#134327	Invalid read	✓	S_regmatch	Fixed	
		5.31.3 (a3c7756)	#134328	Invalid read	✓	S_regmatch	Fixed	
5.31.3 (45f8e7b)	#134342	Invalid read	✓	Perl_mro_isa_changed_in	Confirmed			
MuPDF	539K	1.16.1 (6566de7)	#702253	User after free	✗	fz_drop_band_writer	Fixed	
Boolector	79K	3.2.1 (3249ae0)	#90	NULL pointer dereference	✓	set_last_occurrence_of_symbols	Confirmed	
fontforge	578K	20200314 (1604c74)	#4266	User after free	✓	SFDGetBitmapChar		
		20200314 (1604c74)	#4267	NULL pointer dereference	✓	SFDGetBitmapChar		

WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS

Marcos Tileria
*Royal Holloway,
University of London*

Jorge Blasco
*Royal Holloway,
University of London*

Guillermo Suarez-Tangil
*King's College London
IMDEA Networks*

Abstract

Smartwatches and wearable technology have proliferated in the recent years featured by a seamless integration with a paired smartphone. Many mobile applications now come with a companion app that the mobile OS deploys on the wearable. These execution environments expand the context of mobile applications across more than one device, introducing new security and privacy issues. One such issue is that current information flow analysis techniques can not capture communication between devices. This can lead to undetected privacy leaks when developers use these channels. In this paper, we present WearFlow, a framework that uses static analysis to detect sensitive data flows across mobile and wearable companion apps in Android. WearFlow augments taint analysis capabilities to enable inter-device analysis of apps. WearFlow models proprietary libraries embedded in Google Play Services and instruments the mobile and wearable app to allow for a precise information flow analysis between them. We evaluate WearFlow on a test suite purposely designed to cover different scenarios for the communication Mobile-Wear, which we release as *Wear-Bench*. We also run WearFlow on 3K+ real-world apps and discover privacy violations in popular apps (10M+ downloads).

1 Introduction

Wearable devices are becoming increasingly popular and can now run apps on appliances with large computing, storage, and networking capabilities. According to Gartner, users will spend \$52 billion in wearable in 2020 [14], smartwatches being the most popular gadget. The key feature of these devices is that they are all interconnected, and provide a usable interface to interact with smartphones and cloud-based apps. In Android, wearable devices interact with the smart phone via Wear OS (previously, Android Wear). Wear OS is similar to Android in terms of architecture and frameworks but it is optimized for a wrist experience. Apps in Wear OS can run as standalone programs or companion apps.

Wearable devices provide an additional interface with the digital world, but they are also a potential source of vulnerabilities that increases the attack surface. For instance, a mobile app could access sensitive information and share it with its companion app in another device. Then, the companion app could exfiltrate that information to the Internet. This landscape expands the context of mobile applications across more than one device. Therefore, we cannot assess the security of a mobile app by just looking at the mobile ecosystem in a vacuum. Instead, we need to consider also the wearable app as part of the same execution context.

Previous studies have exposed vulnerabilities on smartwatches and their ecosystem [10, 12, 16, 30]. However, these works have mostly focused on the analysis of wearable apps in isolation [12], their Bluetooth connectivity [16] or the usage of third-party trackers [8]. To systematically study how apps use sensitive data, the security community leverages information flow analysis [3, 9, 15, 18, 21, 32]. Recent works such as COVERT [4], DidFail [18], and DialDroid [6] augment the scope of the data tracking to include inter-app data flows which use inter-component communication (ICC) methods. These works expand the execution context from one mobile app to a set of mobile apps.

In contrast to previous problems, information flow analysis in the wearable ecosystem needs to track sensitive data across apps in different devices, i.e.: the handheld and the wearable. In other words, it needs to consider that data flows can propagate from the mobile app to its companion app (and back) through the wireless connection. In Android, this communication is managed by Google Play Services, a proprietary application which handles aspects like serialization, synchronization, and transmission (among other aspects within the Android ecosystem).

In this work, we present WearFlow, a framework to enable information flow analysis for wearable-enabled applications. To achieve this, we create a model of Google Play Service by leveraging the Wear OS Application Program Interface (API). This enables WearFlow to capture inter-device flows. Thus, we run taint analysis on each app and reason about flows in an

extended context that comprises mobile and companion apps. Our results show that WearFlow can detect Mobile-to-Wear and Wear-to-Mobile data leaks with high precision and finds evidences of misuse in the wild.¹

In summary, we make the following contributions:

1. We propose WearFlow, an open-source tool that uses a set of program analysis techniques to track sensitive data flows across mobile and wearable companion apps. WearFlow includes library modeling, obfuscation-resilient APIs identification, string value analysis, and inter-device data tracking.
2. We develop WearBench, a novel benchmark to analyze Mobile-Wear communications. This test suite contains examples of mobile and wearable apps sharing and exfiltrating sensitive data using wearable APIs as the communication channel.
3. We conduct a large scale analysis of real-world apps. Our analysis reveals that real-world apps use wearable APIs to send sensitive information across devices. Our findings show that developers are not using data sharing APIs following the guidelines given by Google.

The rest of the paper is structured as follows. Section 2 provides an overview of wearable companion apps and Google Play Services. Section 3 presents the security threats of the Mobile-Wear ecosystem. We describe how we model Google Play Services in Section 4. We present WearFlow in Section 5. We evaluate our solution and present the results of our large scale analysis in Section 6. We discuss the limitations of WearFlow and other related works in Sections 7 and 8. Finally, we present our conclusions in Section 9.

2 Background

This section describes the Android-Wear ecosystem, including how Wear applications communicate with their mobile or handheld counter part via Google Play Services. We also provide a motivation example to show the challenges behind tracking data usage in this ecosystem.

2.1 Wearable apps

Wear OS is a stripped version of Android optimized to run wearable apps on Android smartwatches. The capabilities of these smartwatches range depending on the hardware of the manufacturer. Apart from main components such as screen and CPU, these devices incorporate an array of sensors including accelerometers, heart-rate and GPS among others. The Wear OS provides an abstraction for apps to access those sensors.

¹For simplicity, we refer to term Mobile-Wear when we use Mobile-to-Wear and Wear-to-Mobile interchangeably.

Wear OS adopts the same security model used to protect its mobile counterpart. In Android, applications are sandboxed and installed with minimum permissions by default. From Android 6.0, dangerous permissions are not granted at installation-time, but during run-time. Permissions still need to be declared on the app Manifest. The same permission model applies to Wear apps, however the authorization process is independent. This is, permissions are not inherited from the mobile app. The wearable app must request permission to access protected resources. These resources can be either in the smartwatch or in the smartphone (the smartwatch can also access resources in the smartphone and vice-versa provided users grant the appropriate permissions).

Wear devices are also equipped with network connectivity like Bluetooth, NFC, WiFi, or even access to cellular networks. Most watches require a phone pairing process via Bluetooth or WiFi. The pairing process establishes a low-level channel that can be used by mobile apps to communicate with a companion app in the smartwatch. Note, however, that wearable apps can run standalone apps (i.e., no mobile app needed) from Wear OS 2.0. Figure 1 illustrates the interplay between a mobile phone, a smartwatch and the network. We next describe how Wear OS enable apps to communicate with each other (including to how they communicate with the mobile companion app).

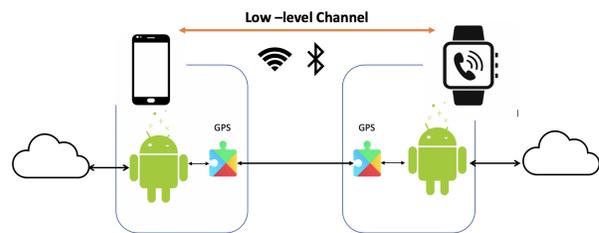


Figure 1: Communication between a mobile app, its companion, Google Play Services (GPS) and the network.

2.2 Google Play Services

While Android is an open-source OS, most “stock” Android devices run proprietary software from manufacturers (OEMs) and third-parties [13]. To access the Google Play Store, Google requires phone manufacturers to include other core modules such as Google Mobile Services (GMS). These services include Google apps (Maps, Youtube, etc.) and background services, also known as Google Play Services.

The Android ecosystem suffered a fragmentation problem as OEMs were unable to keep up with Google updates [33]. In response to the security issues underlying the fragmentation problem, Google moved the most critical components of Android to the Google Play Services bundle. This library receives automated updates from the Play Store without involving OEMs or users. Google Play Services has two core

components: i) a proprietary app that embeds the logic of the different services offered by Google, and ii) a client library that provides an interface to those services. Developers must include the client library in their apps when accessing Google-dependent services, including those regarding Wear. Figure 2 shows how the Google Play Services app interacts with the client library using standard inter-process communication (IPC) channels.

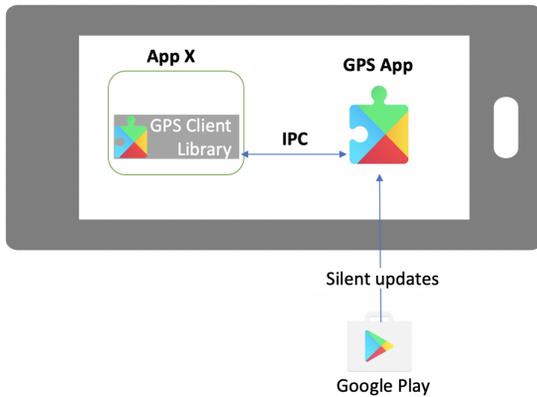


Figure 2: Google Play Services (GPS) architecture and update process.

As of March of 2020, Google provides 19 different packages² that allow developers to interface with all the Google Play Services like Google Analytics, Cloud Messaging, Mobile Ads, or Wear OS among others. In particular, the package *com.google.android.gms.wearable* gathers all the interfaces exposed for wearable apps, including the APIs that enable the communication between mobile and wearables apps. This package is commonly referred as the *Data Layer API*.

2.3 Data Layer

The *Data Layer API* provides IPC capabilities to apps. This API consists of a set of data objects, methods, and listeners that apps can rely on to send data using four types of abstraction:

1. *DataItem* is a key-value style structure that provides automatic synchronisation between devices for payloads up to 100KB. The keys are strings values, and the payload could be integers, strings or other 16 data types. The *DataClient* APIs offers support to send *DataItems* which are uniquely identified by a path (string value) in the system.
2. *Assets* are objects that support large binaries of data like images or audio. *Assets* are encapsulated into *DataItems*

²<https://developers.google.com/android/guides/setup>

before being sent. The *Data Layer* takes care of transferring the data, bandwidth administration, and caching the binaries.

3. *Message* are short bytes of text message that can be used for controlling media players, starting intents on the wearable from the mobile, or request/response communication. The *MessageClient* object provides the APIs to send this type of asynchronous messages. Each message is also identified by a path in the same way as *DataItems*.
4. A *ChannelClient* offers an alternative set of API methods to send large files for media formats like music and video (in streaming as well) which save disk space over *Assets*. *ChannelClient* are also identified by a unique path.

The *Wearable API* also provides the callbacks to listen for events receiving one of these four data types. Table 1 shows a summary of these objects and their corresponding callbacks. We omit the list of API methods due to space constrains. The 16 data types supported by *DataItems* can be found in the API documentation.³

2.4 Mobile-Wear Communication

Once two devices are paired, a mobile and its companion apps can talk to each other through the *Data Layer* as long as they are signed with the same certificate. This is a restriction introduced for security reasons.

Apps can use the *Data Layer* to open synchronous and asynchronous channels over the wireless channel. Table 1 shows the channel type corresponding to each abstraction of the *Data Layer*.

The *MessageClient* (asynchronous API) exposes the methods to put a message into a queue without checking if the message ever reaches its destination. This abstraction encapsulates the context of messages into a single API invocation, for instance, destination and payload. In contrast, synchronous channels (*DataClient*, *ChannelClient*) provide transparent item synchronization across all devices connected to the network. Moreover, synchronous channels rely on many APIs to provide context to one transmission. From now on, we will use synchronous channels to explain the operation of the *Data Layer* as these are more complex than asynchronous.

The context of one transmission consists of: node identifier, channel type (table 1), channel path (string identifier), and the data that will be transferred. Node identifier correspond to string that represents a node in the *Wear OS* network. A channel path represent a unique address which identifies each open channel within a node. Finally the data is the payload of the transmission.

An app can create many channels of the same type to send different payloads to the companion app. Developers often

³<https://tinyurl.com/y4dwoqpk>

Table 1: Map between the different data types and the available channels in the `Data Layer API`.

Data Type - Channel	Channel Type	Information	Listeners
Messages - <code>MessageClient</code>	Asynchronous/not-reliable	Bytes	<code>OnMessageReceived</code>
<code>DataItems</code> - <code>DataClient</code>	Synchronous/reliable	16 types	<code>OnDataChanged</code>
Assets - <code>DataClient</code>	Synchronous/reliable	Binaries	<code>OnDataChanged</code>
Channel - <code>ChannelClient</code>	Synchronous/reliable	Files	<code>OnChannelOpened</code>

use path patterns to create a hierarchy that matches the project structure to identify different channels. For instance, the path `example.message.normal` can be used to request a normal update, while the path `example.message.urgent` could indicate an urgent request.

To initiate a Mobile-Wear communication, the sender app needs to create the context of the channel through a sequence of APIs calls. Then, the Google Play Services app in the phone performs the transmission, handling the encapsulation, serialization, and retransmission (if needed). In the smartwatch, Google Play Services receives the communication and processes the data before handing it over to the wearable app. The receiver app implements a listener that captures events from Google Play Services. The listener could be defined in a background service or an activity where the data is finally processed.

2.5 Motivation Example

In this section, we describe an example of a data leak using the `DataItem` channel. Here, a wearable app sends sensitive information to the Internet after a mobile app transfers sensitive information through this channel. Listing 1 shows the mobile app sending the geolocation and a constant string to the companion wearable app.

First, the channel is created (line 4) with its corresponding path. Then the geolocation and a string “hello” are added to the channel in line 5, and 6 respectively. Finally, the app synchronizes all the aggregated data in one API call (`synchronizeData`) in line 7.

Listing 1: Simplified example of mobile app exfiltrating data to the companion app.

```

1 nodeID = getSmartwatchId()
2 location = getGeolocation()
3 text = "Hello"
4 channel = WearAPI.createChannel("path_x")
5 WearAPI.put(channel, "sensitive", location)
6 WearAPI.put(channel, "greetings", text)
7 WearAPI.synchronizeData(nodeID, channel)

```

Listing 2 shows how the wearable app receives this transmission with the location data.

Listing 2: Example of companion app exfiltrating sensitive data.

```

1 event = WearAPI.getSynchronizationEvent()
2 if (event.path == "path_x"){
3     data = parseEvent(event)
4     location = data["secret"]
5     hello = data["greetings"]
6     exposeToInternet(location)
7     exposeToInternet(hello)
8 } else if (event.path == "path_y"){
9     do_something_else
10 }

```

First, the app fetches the event from the channel using the `Data Layer API` (line 1). The developer uses a conditional statement to execute actions depending on the event’s path. Paths are the only way to characterize events that trigger different data processing strategies when exchanging data through a channel. Here, `path_x` (lines 3 to 7) corresponds to the branch that handles the data sent by listing 1, which includes the geolocation. In this case, the geolocation is sent via a sink in the companion app to the Internet. The branch `path_y` is used to process a different event.

On the receiver side, it also possible to specify the channel path in the Manifest using an intent-filter. In this case, the service only receives events which path is equal to the path specified in the Manifest. However, it also possible to specify a path prefix and then trigger different branches in the code. For listener in activities, developers rely on indirect references to the path on the code, like the example.

3 Security Threats in Wearable Ecosystem

The exchange of sensitive data between mobile and wearable applications introduces risks in relation on how that data may be handled by both the mobile and/or the wearable app. In our case, we assume that the smartphone and smartwatch will contain sensitive information like Personal Identifiable Information (PII), contacts information, and biomedical data that could be exfiltrated either from the smartphone or from the smartwatch.

3.1 Threat Model

We identify the following security risks that arise from the transmission of PII in the mobile-wear ecosystem:

1. **Re-delegation:** The permission model in Android requires developers to declare the permissions of their mobile and wearable apps separately. This enables mobile and wearable apps to engage in colluding behaviors [27]. For instance, a mobile app that requests the `READ_CONTACTS` permission, can use the `Data Layer` APIs to send the contact information to a wearable application that does not have this permission. Similarly, a wearable application could share sensor data such as heart rate or other sensors with its corresponding mobile app without requiring access to the Google Fit permissions.
2. **Wearable data leaks:** Wear OS includes APIs to perform HTTP and other network requests to Internet facing services. This means that wearable apps have exactly the same capabilities to exfiltrate data as regular mobile apps. However, as already mentioned in Section 1, information flow tools available today only account for data leaks that happen directly via the mobile app (or via other apps in the case of collusion). As of today, there are no methods to detect data leaks through wearable interfaces.
3. **Mobile data leaks:** In a similar way, the mobile app could exfiltrate sensitive data leaked from the wearable app environment. An example of sensitive data unique to the wearable is the heart rate. A mobile application can pull this data and sent it over the network. Note that while this threat can be materialized through a permission re-delegation attack, it is not strictly bound to this attack. Instead, both apps can request permission to access specific sensitive data, but the taint is lost when data is transmitted from the companion to the mobile app.
4. **Layout obfuscation:** Developers are increasingly using obfuscation techniques to prevent reverse engineering and to shrink the size of their apps [17]. Obfuscation presents a challenge to information flows analysis when it modifies the signature of relevant classes and methods. In our case, the APIs from the `Data Layer` might be obfuscated, and we cannot merely look at the signatures of the API methods.

To the best of our knowledge, this is the first framework that models Mobile-Wear communication. As a consequence, current frameworks fail to detect the situations above. This happens either when developers are not following good coding practices or when miscreants intentionally try to evade detection mechanism that rely on data-flow analysis.

For simplicity, we do not discuss how permissions are assigned. However, we note that the Motivation Example in

section 2.5 relates to a re-delegation attack when the companion app does not require the geolocation permission. While taint tracking tools are able to identify sensitive data flows in the mobile app, they can not propagate the tracking to the companion app. The simplest way to solve this is to consider the execution context of the mobile (sender) and companion (receiver) app as a single context. Thus, enabling us to reason about existing Mobile-Wear communication and to track non-sensitive message individually.

Note that a Mobile-Wear taint tracking needs to consider that data flows are combined in a single point when the sender transmits the `DataItem`, and it separates again when the receiver app parses the event. This is shown in Listing 1, where the `Data Layer` aggregates data (i.e., the geolocation and a constant value) into a single channel. Finally, we note that an attacker may use any other channel described in Section 2.3 to leak sensitive data, although the technical procedure will defer.

Next, we show how we address this problem for all channels.

4 Modeling Google Play Services

WearFlow expands the context of taint-tracking analysis from a single application to a richer execution environment that includes the wearable ecosystem (i.e., the Wear OS).

Mobile-Wear taint tracking presents a different set of constraints and characteristics than Inter-Application and Inter-Component communication analysis. In Wear OS, the communication between the smartphone and the wearable involves the mobile app, Google Play Services, and the wearable app. As Google Play Services library acts as a bridge between the two, we need to model its behavior to track information between the two apps.

As seen in the examples shown in Listings 1 and 2, wearable APIs are designed to send and receive data in batches. This means that developers first insert the different items they want to transmit between apps and then execute a synchronization API call. From a data analysis perspective, this means that multiple data flows join into a single point when an app invokes the synchronization API to send data. One possible solution would be to taint all the information exchanged. However, this overestimation would result in a high number of false positives. There is another challenge behind tracking individual data flows in Wear OS, i.e.: Google Play Services is not open source and it is implemented in native code, which makes the data tracking more difficult [28].

In order to track these flows, we have created a model of the `Data Layer` to generate a custom implementation of the wearable client library. To create the model, we manually inspected the wearable-APIs from the `Data Layer`, and built a sequence of possible invocations and the effect of these APIs on the context of the communication. This model allows us to extract the context of each communication, such as the

path and the data added into a *channel*. Then, we can use this information to replace the invocations to the original APIs with invocation to our instrumented APIs.

Note that we do not know the details of how Google Play Services implements the communication, but we do know the result of the communication, and we can reason about the context of communication by looking at relevant points where the apps invoke wearable APIs.

The result of our model is a mapping between the original methods from the `Data Layer` client library to a modified implementation template that facilitates the matching of individual data flows between apps. This modified implementation is generated as follows:

1. We identify all relevant classes from the `gms-wearable` library and generate custom signatures for each method.
2. For each app, we identify all invocations of synchronous and asynchronous APIs from the `Data Layer`. For each invocation we run a taint analysis to extract the context of the transmission. This involves:
 - (a) Identifying the channel creation.
 - (b) Searching the items that have been added into the channel variable (data sent across the channel).
 - (c) Evaluating strings from the context (path and keys).
 - (d) Generating custom API calls using the extracted context and corresponding method template.
 - (e) Replacing original method invocation with a custom API invocation.

By doing this, we can simulate the propagation of data flows across apps on different devices while keeping the semantics of the different data flows intact. As an example, whenever we find a call to `<DataClient: putDataItem(PutDataRequest request)>`, the model will tell us that this is a synchronous communication which in sending a `DataItem` (encapsulated in the `PutDataRequest`). In this case, the model also specified that the `PutDataRequest` object required a previous API call that creates a channel, and other APIs that add data to the `DataItem`. We use this information to do a backward and forward inter-procedural analysis to extract such information. Finally, the model provides the rules to match entry and exit points once we have the results of the data flow analysis.

5 WearFlow

We design a pipeline of five phases that result in the detection of Mobile-Wear data leaks. Figure 3 shows a high-level overview of our system. Phase 1 converts the app to a convenient representation and extract relevant information. Phase 2 deobfuscates (if necessary) the Google Play Services client

library and relevant app components. Phase 3 performs a context extraction and instrumentation for every invocation to a wearable API. Phase 4, runs the information flow analysis and export the results. Finally, we match data flows according to the model of the `Data Layer` in phase 5 to obtain all Mobile-Wear flows.

Phase 1: Pre-Processing

Android packs together the wearable and the mobile app into a single package file (namely, APK). `WearFlow` first splits both apps and then uses Soot to pre-process each executable separately. In particular, we convert the Dalvik bytecode into the Jimple Intermediate Representation (IR), and parse the relevant configuration files (e.g., the wearable and mobile app Manifests). Jimple simplifies the different program analysis techniques we use in the following phases.

`WearFlow` then searches for Wear OS components (services and callbacks as in Table 1), subject to an optional deobfuscation phase (Phase 2). We leverage the Manifests to understand the relationship between paths and services by looking at the intent filters declared as `WearableService`. We then inspect the Jimple to obtain all variables of the data types listed in Table 1, including those that appear in callbacks. These data types are used to open Mobile-Wear channels. We will instrument all these components as described in Phase 3.

Phase 2: Deobfuscation

We use a simple heuristic to detect if the app is obfuscated. First, we assume that all Mobile-Wear applications would use any of the methods from the classes of the `Data Layer` API shown in Table 1. Thus, we search for these methods in the client libraries of the APK. If no method is found, we consider the app may be obfuscated and perform a type signature brute-force search. This signature models the type of inputs and outputs of a function.

In addition to the type signature, we further look at local variables declared using system types in the method and compute their frequency per method (when the method is not a stub). The rationale behind including context from the method itself is to reduce the number of false positives when performing the signature search. The signature model uses only system types and abstract types from the `Data Layer` to generate the obfuscation-resilient signatures. We refer the reader to Section 7 for a discussion on our choices and how this may impact our results.

We extract signatures for all relevant methods that model Mobile-Wear IPC (see Section 2.3). Overall we extract 63 signatures capturing methods that exchange `Messages`, `DataItems`, `Assets` and `Channels`. We then search for methods in app's components that match against these signatures. When we find a match, we identify the corresponding wearable API of our interest. As we show in Section 6.4, we can

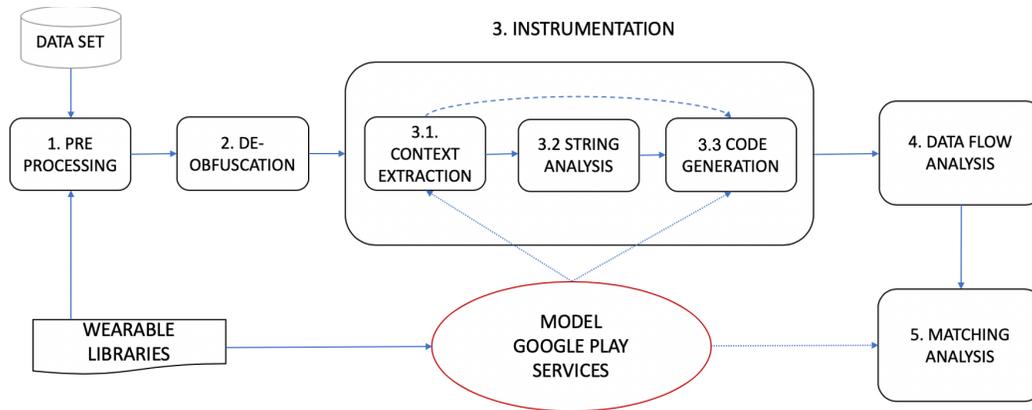


Figure 3: Overview of WearFlow.

identify all the methods used by the model with the above features.

Phase 3: Instrumentation

This phase aims at instrumenting the apps under analysis and it has three steps: context extraction, string analysis, and code generation. It takes as input a model of Google Play Services library. Our attempt to model this library is described in Section 4.

For the context extraction, we search invocations to wearable APIs that send or receive data in each of the components seen in the pre-processing step. Once that one API is identified, WearFlow performs an inter-procedural backward analysis to find the creation of the corresponding channel, and then a forward analysis to find invocations to APIs which add data into the channel. We then evaluate strings of relevant API methods; for instance, the method `<PutDataMapRequest.create(String path)>` for `DataItems`. For this we perform an inter-procedural and context-sensitive string analysis. For asynchronous APIs (`MessageClient`), the context extraction is limited to evaluate the variable which contains the channel path.

The next step is to instrument the app. On the one hand, we add our custom methods to the client library. In particular, we add the declaration of method that we use as entry/exit points in their corresponding classes. On the other hand, for each invocation to APIs methods acting as entry/exit points, we generate the corresponding invocation to our custom APIs. We use the output of the context extraction, string analysis, and the model of the `Data Layer` to generate such invocations. The resulting code will replace the invocations to the original methods in the wearable library.

The Listing 3 shows the instrumented code corresponding to the motivation example in Listing 1. This code replaces the lines [4 - 7] from the example. Note that the code below illustrates a notion of the instrumentation which is done in the Jimple IR.

Listing 3: Simplified instrumented code.

```

1 nodeID = getSmartwatchId ()
2 text = "hello"
3 location = getGeolocation ()
4 channel = WearAPI.createChannel ("path_x")
5 WearAPI.syncString (nodeID, channel,
6   "greetings", text)
7 WearAPI.syncString (nodeID, channel,
8   "sensitive", location)

```

Phase 4: Data Flow Analysis

This phase performs data flow analysis of the Mobile-Wear ecosystem as a whole. First, we add callbacks from the Wear OS libraries as given by our model to enable data flow analysis across devices (see Section 2.3). Note that we add sources and sinks that are not detected by state-of-the-art well-maintained projects in Android [2, 3]. More importantly, we add data wrappers that can capture how data flows propagate through objects of the `Data Layer`. One limitation of existing data flow frameworks like `FlowDroid` [3] is that they use simplified wrapper models that only abstract the semantics of the Android framework for well-known cases.

The next step is to compute the call graph of both apps and perform a taint tracking analysis as a single context. We do this by first running the taint analysis separately on each app and then matching the results using the instrumented APIs as connectors between data flows. We add the APIs that send data as sinks (wearable-sinks) and the APIs that receive data as source (wearable-sources). Then, we also add the wearable-sources and the wearable-sinks in the list of sources and sinks.

Finally, WearFlow reports the results of the taint tracking. At this point, we are only interested in data flows with wearable-sources or wearable-sinks. It is worth noting that taint analysis still detects data flows that end in a non-wearable

sink, but they are irrelevant to the matching step. Our approach is agnostic to the underlying method used to compute data flows. We refer to Section 6.1 for implementation details.

Phase 5: Matching Analysis

The final step consists of matching exit points with entry points; that is to say, wearable-sinks with wearable-sources. We consider three values to match data flows: channel path, API method, and key. If the value of the path or key could not be calculated during the context extraction, then we use a wildcard value that matches any value. To match the API methods, we built a semantic table that provides information to match wearable-sinks with its corresponding wearable-sources. We present a summary in the Table 2 due to space limitation. The table contains thirty-four entries in total and it can be found in the project repository.

6 Evaluation and Results

We evaluate WearFlow against other Android information flow analysis tools currently available and perform a large-scale analysis of 3.1K Android APKs with wearable components looking for sensitive data leaks. Our evaluation uses a specifically crafted set of apps that presents different data exfiltration cases using the `Data Layer` API. We conduct our experiments on a machine with 24 cores Intel Xeon CPU E5-2697 v3 @ 2.60GHz and 32 GB of memory.

6.1 Implementation

WearFlow relies on the Soot framework [29] to perform the de-obfuscation, context extraction and app instrumentation (Phases 2, 3.1 and 3.3). Our implementation leverages FlowDroid [3] with a timeout of 8 minutes per app for the information flow analysis (Phase 4) and Violist [19] for the string analysis (Phase 3.2). We use FlowDroid and Soot because previous works report that they provide a good balance between accuracy and performance on real-world apps [6, 24, 25]. We customize FlowDroid to run on wearable apps by adding callbacks from the Wear OS libraries and by extending the SuSi [2] sources and sinks as discussed in Section 5. We also perform several optimizations to Violist to reduce the execution time while keeping the accuracy for the APIs we were interested in. For instance, we reuse the control flow graph generated by Soot, and we limit the evaluation of the strings to relevant methods. With this, WearFlow adds, overall, around 6000 LoC to these frameworks. We make the implementation of WearFlow open source in <https://gitlab.com/s3lab-rhul/wearflow/>.

6.2 Evaluation results

As community lacks on a test suite that include Mobile-Wear information flows for Android, we create WearBench⁴. WearBench has 15 Android apps with 23 information flows between the mobile app and the wearable companion (18 of them sensitive). Our test suite covers examples of all APIs from the `Data Layer`. It also contains challenges for the instrumentation like field sensitivity, object sensitivity, and branch sensitivity for listeners.

Our suite is inspired by Droid-Bench⁵ and ICC-Bench⁶, which are standard benchmarks to evaluate data flow tools. Note that these benchmarks evaluate the effectiveness of the taint analysis, and some Inter-App communication cases using ICC methods. Instead, we are evaluating Inter-App communication between mobile and wearable apps (using the `Data Layer` API). Therefore, we cannot use these benchmarks alone to evaluate WearFlow. In our evaluation, we compare our results against FlowDroid. For this, we add the `Data Layer` APIs as sources and sinks, execute FlowDroid on both the mobile and wearable companion and look for matches. We run FlowDroid with a context sensitive algorithm twice: first with high precision, we set the access path length to 3. Then we reduce the precision by setting the access path to 1. With this, FlowDroid truncates taints at level 1. This configuration increases the number of false positives but catches situations where FlowDroid fails to propagate taint abstraction correctly.

Table 3a shows the result of our evaluation against the test suite. WearFlow detects all the 18 exfiltration attempts with two false positives. These two false positives stem from a branching sensitivity issue present in FlowDroid, which result in false positives during the data flow analysis (Phase 4). Conversely, FlowDroid with high precision detects 6 out of 18 exfiltrations — these are only matches communicating with the `MessageClient` API. This is because FlowDroid fails to propagate taints on complex objects from the `Data Layer`. When reducing the precision, FlowDroid identifies matches with `MessageClient` and `DataClient` but still fails to identify sensitive flows with the `ChannelClient` API. In this case, FlowDroid produces 12 false positives. This results from an overestimation of taints that uses `DataItem`.

Our results show that WearFlow performs better than FlowDroid by a clear margin. This exemplifies how the modeling, instrumentation and matching analysis can improve information flow analysis in wearable applications.

6.3 Analysis of Real-World Apps

We use WearFlow to search the presence of potential data leaks on around 3.1K real-world APKs available in the Google

⁴<https://gitlab.com/s3lab-rhul/wearbench/>

⁵<https://github.com/secure-software-engineering/DroidBench>

⁶<https://github.com/fgwei/ICC-Bench>

Table 2: Selection of Sink-Source matches in Data Layer API. A full list can be found in WearFlow repository.

Library	Wearable sink signature	Wearable source signature
DataClient	DataClient: void putString(String,String)	DataMap: String getString(String)
MessageClient	MessageClient: Task sendMessage(String,String,byte[])	MessageEvent: byte[] getData()
DataClient	DataClient: void putAsset(String,Asset)	DataMap: Asset getAsset(String)
ChannelClient	ChannelClient sendFile(Channel,Uri)	Task receiveFile(Channel,Uri,Boolean,String)

Table 3: Summary of our results.

(a) Results for our test-suite between WearFlow and FlowDroid. HP = high precision, LP = low precision.

Library	Existing Data Flows			Found Data Flows		
	Apps	Total	Sensitive	WearFlow	Flowdroid-HP	Flowdroid-LP
DataItem	9	16	13	14 (1 FP, 1 FN)	0 (13 FN)	22 (6 FP)
Message	5	6	4	5 (1 FP)	6	10 (6 FP)
Channel	1	1	1	1	0 (1 FN)	0 (1 FN)

(b) Results for real-world apps (* sensitive data flows).

	Apps	APKs
Number of apps	220	3,111
With flows	47	293
With sensitive *	6	50

Play Store (downloaded from AndroZoo [1]). From an initial set of 8K APKs, around 5K refer to standalone (only wear) APKs, and 3.1K include mobile and wearable components. We execute WearFlow against this set which corresponds to 220 different package names. Table 3b shows a summary of the results. Note that the dataset contains multiple versions of the same app. Thus, we refer to apps as APKs with unique package name.

Figure 4 shows a summary of the different APIs used as exit/entry points of sensitive data flows. Although we found the occurrence of the ChannelClient API in the dataset, we did not find any case where this API was used to send sensitive information. WearFlow identifies sensitive information flows that include the transmission of device contacts (via Cursor objects), location, activities, and HTTP traffic. We also found that in several occasions sensitive data ended up in the device logs (17% of overall sinks) or SharedPreferences files (20%). A more detailed analysis of these flows for a selection of apps is provided in Section 6.5.

WearFlow is capable of finding 4,896 relevant data flows in all the analyzed APKs. Out of those, 388 relate to Mobile-Wear sensitive information flows in 6 apps (or 50 APKs, when considering all versions and platforms). The results indicate that 70% of the flows are from the mobile to the wear platform, while 30% are wear to mobile.

6.4 Applicability

We next see how we perform when dealing with obfuscation and what is the runtime overhead.

Obfuscation. WearFlow detects 282 obfuscated APKs in the dataset. The deobfuscation phase successfully unmangles all these APKs. On the one hand, we find 71 data flows using

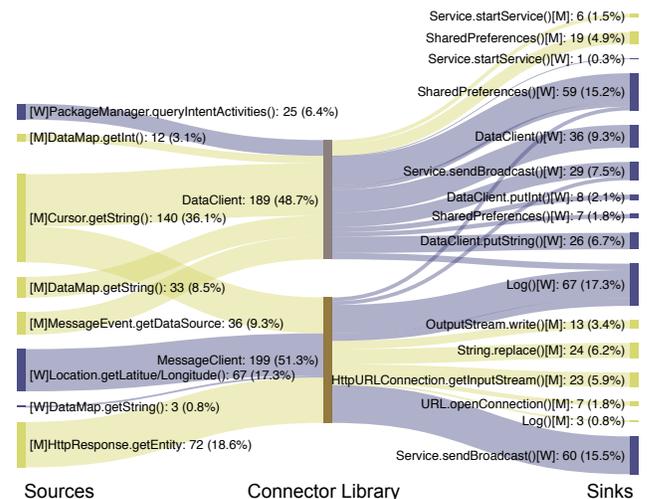


Figure 4: Sensitive information flows found. [M] refers to Android and [W] refers to Wear OS.

the Data Layer within these APKs. WearFlow did not find relevant APIs in 651 APKs. This can either because these APIs are not used at all or because developers use more complex obfuscation techniques. We discuss this in Section 7.

On the other hand, we find around 2K non-obfuscated APKs in our dataset. WearFlow instruments 4.8 components on average per APK (excluding library classes). From all the wearable APIs, around 48% are DataClient APIs, 51% MessageClient APIs, and less than 1% ChannelClient APIs. This number shows that developers are aggregating multiple data into DataClient before synchronizing DataItems and shows the benefits of instrumenting the APKs to track individual data flows.

Running time. Running our tool on the real-world dataset took 115 hours. WearFlow analyzes over 95% of the APKs before the 8 minutes timeout lapses. The average time per APK is 3.1 minutes. Note that wearable apps are considerably smaller in size than mobile apps, and WearFlow evaluates most wearable apps in less than 1 minute. The time distribution per phase analysis is the following: pre-processing 13%, string analysis 9%, deobfuscation and instrumentation 2%, and data flow analysis 76%.

WearFlow failed to complete the analysis for a small number of APKs. In most cases, this is due to unexpected bytecode that Soot fails to handle, errors while parsing APK resources, or because the analysis reached an extended timeout.

Overall, WearFlow extracts data flows for an additional of 282 Mobile-Wear APKs. Without the deobfuscation phase, these flows would not otherwise be extracted. The deobfuscation phase only takes 2% of the running time.

6.5 Case Studies

This section describes issues found by WearFlow in specific apps in relation to the threat model presented in Section 3.

Companion Leak. We first study the case of Wego (*com.wego.android*), a travel app to book flights and hotels with more than 10 million downloads. We find a sensitive flow that starts in the watch with source `getLatitude()` from the Location API, and it is sent to the mobile app with the `MessageClient` API using the path “request-network-flights”. Then, the mobile app sends out this data through URL using the `URLConnection` object, and write it to a file system using the `java.io.OutputStream` class. In this case, both the wearable and the mobile declare the location permission in the Manifest. However, this alone is not enough to comply with the guidelines.⁷ In this case, the wear app must send the user to the phone to accept the permission. This case shows that it is possible to bypass the permission system using `Data Layer` APIs.

Permission re-delegation. *Venom* (*fr.thema.wear.watch-venom*) is a Watch Face customized for a watch user interface. The mobile version of this app uses the `android.database.Cursor` class to store sensitive information such as the call history or unread messages in a database. The app aggregates all information in a `DataItem` object and synchronizes it with the wearable app. However, the wearable app does not declare the relevant permissions. Interestingly, the string analysis has been key to uncover the type of information the app is retrieving from the database and trace it

⁷<https://developer.android.com/training/articles/wear-permissions>

back to API sources that relate to the sensitive information discussed (e.g., missed calls and text messages).

Sensitive Data Exposed Finally, we observe evidence of apps exposing sensitive data through a wide range of sinks, including Android Broadcast system and Shared Preferences. For instance, Talent (*il.talent.parking*) — which is used for car parking — reads data related to the last parking place and its duration from a database and synchronizes it with the watch using a `DataItem`. Then the wearable app writes the data to Shared Preferences. Another example is the app *com.mobispector.bustimes*, which shows bus and tram timetables, and has more than 4 million downloads. The app reads data from an HTTP response, then send it to the wearable through the `MessageClient` API, and finally executes a system Broadcast exposing the content of the HTTP response.

All these cases show how developers leverage the `Data Layer` API to send sensitive information. While it is unclear whether or not these cases intentionally use Google Play services to hinder the detection of data leakages, we see that WearFlow is effective at exposing bad practices that can pose a threat to security and privacy.

7 Limitations

This section outlines the limitations of our work. These may arise from WearFlow’s implementation or the dataset we used.

Data Transfer Mechanisms. WearFlow inherits the limitations of static analysis, i.e.: it is subject to constraints of the underlying flow and string analysis techniques. This means that it fails to match data flows with native code, advanced reflection, or dynamic code loading. Still, WearFlow can be used together with other frameworks [22, 31] that handle these issues to improve the accuracy of the analysis.

WearFlow considers obfuscation while performing the analysis of apps. There are four trivial techniques and seven non-trivial techniques commonly used in the wild, according to a large scale study of obfuscation in Android [17]. WearFlow type-signature deobfuscator is resilient to all trivial techniques and five non-trivial, but it fails to deobfuscate APKs with class or package renamed and reflection. As mentioned before, WearFlow did not found relevant APIs in 20% (651) of the APKs, many of which correspond to APKs with these obfuscation techniques.

A more robust deobfuscation technique should rely on other invariant transformations such as the hierarchical structure of the classes and packages. Many methods of the `Data Layer` library are stubs, therefore using features of the method body will not improve the accuracy drastically. Additionally, one could obfuscate those features (e.g., using reflection together with string encryption) introducing false negatives.

In our case, we have successfully leveraged system types to disambiguate methods with the same signature. We chose to only use system types as they are less prone to obfuscation than other data types and have helped reducing the number of false positives. One could take into account the threat model, and chose to use a more coarse type of signatures when certain traits (e.g., reflection) appear in the app.

Branching. Another potential source of false negative data flows stems from the backward analysis, in the context extraction. WearFlow stops the backtracking when encounters a definition of a channel, but this definition could be part of the branch of a conditional statement. Depending on the scope of the variable, the channel could be defined in another method or even another component. Finally, if the string analysis is unable to calculate the value of a key or path (e.g., when there are multiples values due to branching) we use a wildcard value, i.e.: we match any string. This means that the matching step will overestimate the potential flows between the entry and the exit point.

Dataset. Our dataset is limited to 3,111 APKs and 220 package names after considering different APK versions. There are more than 220 apps available for Wear OS, however, identifying them is a challenging task. Google Play does not offer an exhaustive list of apps with Wear OS components, nor it is always featured in the description of the app. This restriction limits our ability to query Wear OS apps in Google Play. Furthermore, datasets like Androzoo do not provide information about whether a app has wearable components or not. Thus, we need to download apps as the rate limit allows. Given the low density of these kind of apps in the overall set of Android apps, the amount of apps that can be obtained this way is very limited.

Model Accuracy. The precision of the analysis also depends on the accuracy of the `Data Layer` model. The `Data Layer` model used by WearFlow replicates the `Data Layer` model, as described in Google’s Wear OS documentation. If the `Data Layer` APIs were to transfer data through undocumented components of the OS or even through the cloud (e.g., via backups), WearFlow would not detect such flows. Also, our model is based on Wear OS versions 1 and 2. Wear OS under active development. Thus, any new APIs introduced in future versions will need to be modeled.

8 Related Work

Mobile-Wear communication can be seen as a kind of inter-app communication where one of the apps is being executed in a wearable device. Several works have focused on app collusion detection [4, 5, 18, 23, 26, 34]. These works model ICC methods to identify sensitive data flows between applications

running on the same device. WearFlow complements these, extending the analysis of these apps into the Mobile-Wear ecosystem, increasing the overall coverage of these solutions to all current app interactions in the Android-Wear OS ecosystem. One may argue that these tools could be extended to cover for Wear OS interactions. As an example, works such as DialDroid [6] uses entry and exit points to match ICC communication between mobile apps. In our case, we consider wearable APIs as sources and sinks, which could be easily replicated in DialDroid. However, These APIs aggregate multiple data into a single API call, and we need to match data types on the sender and receiver side, which would lead to inaccuracies in DialDroid and many other tools [11].

ApkCombiner [20] combines two apps into one allowing to run taint tracking on a single app. This approach does not allow us to reason about individual items aggregated into a single API call.

There has been a recent interest of the community in expanding the scope of data tracking to more platforms outside the Android ecosystem. Zou *et al.* [35] studied the interaction of mobile apps, IoT devices and clouds on smart homes using a combination of traffic collection and static analysis. They discovered several new vulnerabilities and attacks against smart home platforms. Berkay *et al* proposed a taint tracking system for IoT devices [7]. WearFlow could have been implemented following the same approach (analysing WiFi and Bluetooth communications between Android and Wear OS). This would have required us to reverse the different communication protocols and data exchanged in both wireless protocols. Our approach is simpler, and doesn’t require additional hardware to execute.

9 Conclusion

In this work, we have presented WearFlow, a static analysis tool that systematically detects the exfiltration of sensitive data across the Mobile-Wear Android ecosystem. WearFlow augments the capabilities of previous works on taint tracking, expanding the scope of the security analysis from mobile apps to smartwatches. We addressed the challenge of enabling inter-device analysis by modeling Google Play Services, a proprietary library. Our analysis framework can deal with trivial obfuscation and most of the non-trivial obfuscation techniques commonly used in the wild.

We have created WearBench, the first benchmark for analyzing inter-device data leakage in Wear OS. Our evaluation shows the effectiveness of WearFlow over other approaches. We also analyze apps in Google Play. Our results show that our system scales and can uncover privacy violations on popular apps, including one with over 10 million downloads. As a future work, we want to extend our deobfuscation phase to cover additional forms of obfuscation (e.g., the two — out of seven — non-trivial obfuscations we discuss), and extend the scope of our analysis to the entire Google Play app market.

10 Acknowledgements

This research has been partially sponsored by the Engineering and Physical Sciences Research Council (EPSRC) and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

References

- [1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16*, page 468–471, New York, NY, USA, 2016. Association for Computing Machinery.
- [2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013*, 114:108, 2013.
- [3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.
- [4] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE transactions on Software Engineering*, 41(9):866–886, 2015.
- [5] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers Security*, 70:392–421, 2017.
- [6] Amiangshu Bosu, Fang Liu, Danfeng Daphne Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.
- [7] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium*, pages 1687–1704, 2018.
- [8] Jagmohan Chauhan, Suranga Seneviratne, Mohamed Ali Kaafar, Anirban Mahanti, and Aruna Seneviratne. Characterization of early smartwatch apps. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.
- [9] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–12, 2015.
- [10] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience*, 47(3):391–403, 2017.
- [11] Karim O Elish, Danfeng Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. In *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*, 2015.
- [12] HP Fortify. Internet of things security study: smartwatches. Accessed March 2020, 2015. https://www.ftc.gov/system/files/documentspublic_comments/2015/10/00050-98093.pdf.
- [13] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. *arXiv preprint arXiv:1905.02713*, 2019.
- [14] Gartner. Gartner says global end-user spending on wearable devices to total \$52 billion in 2020. Accessed March 2020, 10 2019. <https://perma.cc/MR8J-PUUK>.
- [15] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.
- [16] Rohit Goyal, Nicola Dragoni, and Angelo Spognardi. Mind the tracker you wear: a security analysis of wearable health trackers. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 131–136, 2016.
- [17] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering*, pages 421–431, 2018.

- [18] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.
- [19] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 661–672. ACM, 2015.
- [20] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*, pages 513–527. Springer, 2015.
- [21] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ocateau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.
- [22] Li Li, Tegawendé F Bissyandé, Damien Ocateau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 318–329, 2016.
- [23] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Yao, Karim O Elish, and Barbara G Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. In *2017 IEEE Security and Privacy Workshops (SPW)*, pages 189–198. IEEE, 2017.
- [24] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341, 2018.
- [25] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droid-safe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.
- [26] Tristan Ravitch, E Rogan Creswick, Aaron Tomb, Adam Foltzer, Trevor Elliott, and Ledah Casburn. Multi-app security analysis with fuse: Statically detecting android app collusion. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pages 1–10, 2014.
- [27] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.
- [28] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342, 2016.
- [29] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.
- [30] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. Mole: Motion leaks through smartwatch sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 155–166, 2015.
- [31] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.
- [32] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.
- [33] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.
- [34] Mengwei Xu, Yun Ma, Xuanzhe Liu, Felix Xiaozhu Lin, and Yunxin Liu. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web*, pages 143–152, 2017.
- [35] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th {USENIX} Security Symposium*, pages 1133–1150, 2019.

MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing

Yaohui Chen
Northeastern University
yaohway@gmail.com

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Reza Mirzazade farkhani
Northeastern University
reza699@ccs.neu.edu

Boyu Wang
Stony Brook University
boywang@cs.stonybrook.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Seed scheduling highly impacts the yields of hybrid fuzzing. Existing hybrid fuzzers schedule seeds based on fixed heuristics that aim to predict input utilities. However, such heuristics are not generalizable as there exists no one-size-fits-all rule applicable to different programs. They may work well on the programs from which they were derived, but not others.

To overcome this problem, we design a Machine learning-Enhanced hybrid fUZZing system (MEUZZ), which employs supervised machine learning for adaptive and generalizable seed scheduling. MEUZZ determines which new seeds are expected to produce better fuzzing yields based on the knowledge learned from past seed scheduling decisions made on the same or similar programs. MEUZZ extracts a series of features for learning via code reachability and dynamic analysis, which incurs negligible runtime overhead (in microseconds). MEUZZ automatically infers the data labels by evaluating the fuzzing performance of each selected seed. As a result, MEUZZ is generally applicable to, and performs well on, various kinds of programs.

Our evaluation shows MEUZZ significantly outperforms the state-of-the-art grey-box and hybrid fuzzers, achieving 27.1% more code coverage than QSYM. The learned models are reusable and transferable, which boosts fuzzing performance by 7.1% on average and improves 68% of the 56 cross-program fuzzing campaigns. When fuzzing 8 well-tested programs under the same configurations as used in previous work, MEUZZ discovered 47 deeply hidden and previously unknown bugs, among which 21 were confirmed and fixed by the developers.

1 Introduction

Hybrid testing as a research topic has attracted tremendous attention and made significant contributions to bug discovery. For instance, the winning teams in the DARPA Cyber Grand Challenge [6] all used hybrid testing [17]. Compared with plain fuzzing, hybrid testing features an extra concolic execution component, which revisits the fuzzed paths, solves the path conditions, and tries to uncover new paths.

One key challenge in hybrid testing is to recognize high-utility seeds (*i.e.*, seeds of high potential to guide concolic execution to crack complex conditions guarding more coverage and bugs). Prioritizing such seeds allows the hybrid fuzzer to achieve higher code coverage more quickly, and in turn, discover more bugs in a fixed time frame. Moreover, this prioritization matters in practice because the concolic execution engine usually has limited time budget and can explore only a (small) subset of all fuzzer-generated seeds. Being able to estimate seed utility allows hybrid fuzzers to use concolic execution more efficiently.

The existing work [8, 19, 25, 33, 49, 54, 55] uses purely heuristic-based seed selection. For example, some prefer seeds with smaller sizes while some value those that lead to new code coverage. These heuristics, despite their simplicity, do not perform equally well across different kinds of programs and are not universally suitable for all programs. Contradicting the previous belief [8, 13, 33], our experiments show that seeds leading to new coverage sometimes have the lowest utility (§6.3). Similarly, previous work [33, 54] suggested that smaller seeds should have higher utility, which however is not true in certain programs as our evaluation shows. As a result, these simple and fixed heuristics may cause non-optimum seed selections, overwhelming the concolic engine with low-utility seeds and slowing down bug discovery.

Compared to heuristics, Machine Learning (ML) algorithms, when trained with sufficient data, can discover complex and implicit patterns automatically [43]. We show that seed selection strategies that are automatically learned based on individual programs perform better than manually defined heuristics that fail to consider all kinds of programs. As our experiment shows that the influence of each feature varies across different programs, suggesting that no single feature (or rule) can work well for all programs. ML-based seed selection avoids the need for manually designing, testing, and reasoning about seed selection rules, which can be daunting, non-scalable, or even impossible when the volume of data to be analyzed is overwhelming.

In this paper, we introduce MEUZZ, an ML-enhanced hybrid fuzzing system. Unlike existing work, which schedule

seeds using simple heuristics derived from a particular set of test programs, MEUZZ uses ML and a set of static and dynamic features, computed from seeds and individual programs, to predict seed utility and perform seed scheduling. MEUZZ also has a built-in evaluation module that measures prediction quality for continuous learning and improvement. To the best of our knowledge, MEUZZ is the first work [42] that applies ML to seed prioritization and scheduling.

To effectively apply ML to seed scheduling for hybrid fuzzing, our design of MEUZZ pays special attention to two ML tasks: *feature engineering* and *data labeling*. While these are the essential steps to bootstrap ML, they could be time-consuming and thus too costly or impractical to be included in the fuzzing workflow. For instance, feature extraction can be very slow if it requires heavy computation or extensive data collection. Moreover, it is not straightforward to quantify seed utility, which is essential for labeling.

To tackle the aforementioned challenges, we first engineer a set of lightweight features based on code reachability and dynamic analysis. Second, we propose a labeling method using the input descendant tree to quantify the utility of a seed. Our evaluation shows that MEUZZ takes only $5\mu s$ on average to extract an individual feature. It also confirms that the descendant tree of a seed accurately reflects seed utility.

Collecting data and training a new model for every program might not be economical or necessary. Therefore, we investigate the feasibility of model reusability and transferability to answer the question: *Is a learned model transferable to different fuzzing configurations or programs?* Since the learning is designed to predict the likelihood of seeds triggering bugs, rather than any specifics of the fuzzed program, a model learned by MEUZZ turns out to be applicable beyond the program from which the model is learned.

We compare MEUZZ with the state-of-the-art fuzzers [19, 24, 33] as well as the most recent hybrid testing systems [25, 54]. The results, based on a set of real-world benchmark programs, show that MEUZZ achieves much higher code coverage than the tested fuzzers that use simple seed selection heuristics. Particularly MEUZZ expands the code coverage by as much as 27.1% compared to QSYM, the start-of-the-art hybrid fuzzing system. The experiments also show that the prediction models learned by MEUZZ have good reusability and transferability. The reused models boost the coverage by 7.1% on average. The transplanted models improve fuzzing performance in 38 out of 56 cases (67.9% of cases), with 10 cases seeing more than 10% improvement.

This paper makes the following contributions.

- *Effective and generalizable approach.* We design, implement, and evaluate MEUZZ, the first system that applies machine learning to the seed selection stage of hybrid fuzzing. MEUZZ performs better and is more widely applicable than heuristic-based seed selection.
- *Practical feature and label engineering.* We address two major challenges, namely feature engineering and label

inference, when applying ML to seed selection in hybrid fuzzing. Our feature selection and extraction allow for online/continuous learning. They are compatible with the existing hybrid fuzzing workflow and require no changes to either fuzzers or concolic execution engines. We also propose an automatic label inference method based on seed descendant trees.

- *Reusable and transferable ML models.* Our seed selection models demonstrate strong *reusability* and *transferability*. As a result, MEUZZ can reuse a well-trained model on different programs (or different fuzzing configurations) to quickly bootstrap the fuzzing campaign and continuously improve and adapt the model to the current program or configuration.
- *Open-Source.* The full implementation of MEUZZ will be open-sourced after acceptance.

2 Motivation

The seed selection (or scheduling) in fuzzing aims to solve this problem: given a program and a set of seeds, in which order the fuzzer should test the seeds to maximize the gain during a fixed period. Seed selection plays a critical role in hybrid fuzzing because the concolic execution engine can only explore an (often small) subset of the seeds due to time constraints. Hence, hybrid fuzzing cannot fully benefit from concolic execution if the seed selection is not optimal.

Why seed selection is important for hybrid fuzzing:

Hybrid fuzzers without a seed scheduling mechanism (*e.g.*, Driller [49]) have to explore all inputs. This “brute force” strategy has two main drawbacks. First, concolic engines cannot keep up with the speed of plain fuzzing because they run relatively slowly and often encounter path explosions and timeouts. As an experiment, we used QSYM [54] to fuzz a set of real-world benchmark programs. QSYM is one of the state-of-the-art concolic execution engines for hybrid testing¹. As shown in Figure 1, for a continuous 24-hour run, QSYM was only able to explore 23.1% of the seeds in fuzzer’s queue.

Second, a seed selection strategy affects fuzzing results drastically. A naive strategy delays a fuzzer’s exploration of interesting program locations, and sometimes, prematurely forces the fuzzer to skip deep program paths and states. Some recent research [19, 25, 51, 54, 55] studied a few seed selection heuristics of various levels of sophistication. In their experiments, fuzzers using these seed selection heuristics produce better results (*e.g.*, higher code coverage) than fuzzers with naive or no strategies.

Why exploring machine learning for seed selection:

All the existing seed selection strategies are based on manually defined heuristics. Although performing well on their selected benchmarks, these strategies may not be generalizable to, or suitable for, other programs. For instance, DigFuzz [55] and AFLFast [19] prioritize seeds with less explored paths

¹Reportedly, QSYM is 3x faster than Driller [54].

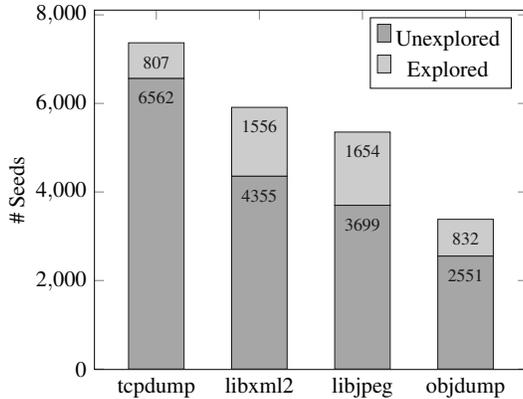


Figure 1: The total number of inputs explored by the concolic execution engine of QSYM in 24 hours. On average, only 23.1% of the inputs were explored by the concolic execution even though the engine was continuously running.

by fuzzer. Savior [25] prefers seeds dominating more UBSan-labeled code paths. QSYM [13] prioritizes seeds with smaller sizes. These heuristics are all based on intuition or empirical observations gained from limited test cases or benchmarks.

A biased or unsuitable seed selection strategy delays or prevents fuzzers’ exploration of deep program states or the discovery of bugs. For instance, QSYM [13] and ProFuzzer [53] prioritize inputs with smaller sizes. Their developers observed in their evaluation benchmarks that smaller inputs lead to higher code coverage. However, as [25] pointed out, QSYM fails to explore a large chunk of code in program who (a program in the LAVA-M benchmark [27]) due to the unsuitable seed selection strategy (*i.e.*, only inputs larger than a certain size can trigger the vulnerable functions in this case).

This clearly indicates that fixed seed selection heuristics can hardly be suitable for a wide set of programs (See Figure 10).

Due to the diverse scheduling scenarios, modern fuzzers (*e.g.*, AFL [2], QSYM [13]) often employ multiple heuristics for seed prioritization. Unfortunately, relying on human efforts to learn and generalize seed selection strategies, as the previous work did, is not scalable to a large number of features. In fact, it is just infeasible to manually reason about a big set of selection criteria when the number of features and the amount of data to be analyzed become overwhelming (*e.g.*, OSS-Fuzz generates four trillion seeds per week [4] for different programs).

In contrast to heuristics, machine learning (ML) is good at discovering underlying connections between data attributes [36, 43]. ML can be applied to seed selection because, as shown by existing studies, the selection strategies are indeed learnable (*i.e.*, exhibiting statistically significant patterns). With sufficient learning data, ML can not only infer the importance of different features but also mine the integration rules at scale.

MEUZZ is the first to explore the ML-based, data-driven approach to seed selection in hybrid fuzzing. Our result confirms that automatically and continuously learned seed selection strategies are more suitable for individual programs.

3 Background

Hybrid fuzzing [25, 49, 54] combines fuzzing and concolic execution to address the deficiencies of both the approaches. Figure 2 shows an overview of a general hybrid fuzzing framework. The whole system consists of three major components: fuzzer, concolic testing, and coordinator. For the sake of brevity, we refer the interested readers to [2, 8, 23, 29] for the technical details of fuzzing and concolic execution.

3.1 Hybrid Fuzzing

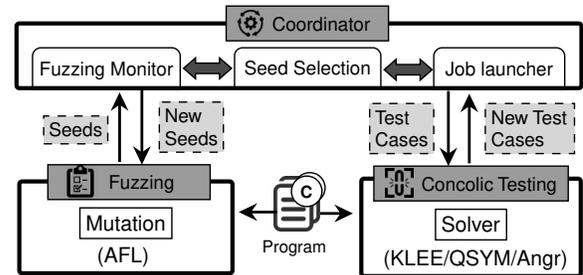


Figure 2: General hybrid fuzzing workflow.

We dissect the coordinator component as it is less discussed in the literature and is the focus of this work. The coordinator is a middleware that regulates the other two components. Its major tasks include (*i*) monitoring the fuzzer to decide when to launch the concolic execution engine, and (*ii*) prepare the running environments for concolic testing; and (*iii*) select and filter inputs that flow between fuzzer and concolic executor.

The seed selection module in the coordinator needs to decide which seeds in the fuzzer’s queue should be transferred to the concolic testing first (*i.e.*, Seed utility prediction phase). Before launching the concolic execution, the coordinator needs to rank all inputs in the fuzzer’s queue based on their utility. The *utility* of seed should correspond to the estimation of its power to produce additional coverage if it is selected to fuzz. As we mentioned in Section 2, current methods use various heuristics to achieve this prioritization goal.

3.2 Supervised Machine Learning

Supervised ML is the task of learning from labeled data and applies the knowledge to unknown data. Classification and regression are two foremost categories of such algorithms. While classification is used for predicting categorical responses, regression predicts a numerical value to the new data based on previously observed data. Supervised learning has shown thriving employment in application security, including bug discovery [30, 35, 37].

Supervised machine learning can be either online or offline. The difference between these two lies in how models are updated.

Online learning: Some learning environments can change from second to second and their models need to get updated

(or relearned) as fast as they see a new sample. Under this constraint of time, online learning shows promises by only considering the new data to update the model, which makes it an efficient approach. Basically, most learning algorithms that are compatible (but not limited) with the standard optimization algorithms like stochastic gradient descent (SGD) can learn incrementally.

Offline learning: In contrast to online learning, the models in offline learning need to be retrained with the whole dataset as newer data appear. One of the successful examples of offline supervised learning techniques is Random Forest (RF), which has shown promising achievements, and in certain domains, has even better performance than neural networks [28]. In addition to RF, deep learning has been shown success in different domains; however, they are usually practiced on unstructured data such as images and they require a relatively larger amount of data to perform well [16]. Moreover, such techniques need high computational power and longer time to train; hence they are not suitable for the online fuzzing workflow.

4 System Design

4.1 System Overview

MEUZZ is the first machine learning-based hybrid fuzzer that learns from the previously observed seeds and identifies which kinds of seeds have the potentials to more effectively explore the program being tested.

Figure 3 shows an overview of MEUZZ. MEUZZ starts fuzzing (1) a program with pre-defined or empty seeds. It then extracts features (2) from the program as well as the seeds (§4.3) to model coverage gains. Such features are used to predict (3) the coverage that unknown seeds may provide (§4.5). Concolic engine (4) then receives the potentially influential seeds from the prior step and produces mutated seeds. Next, MEUZZ guides the fuzzer to use these seeds and their generated mutants—by the evolutionary algorithms—to continually test the program. In the beginning, the prediction model is randomly initialized, so the prediction quality is uncertain. But as fuzzing continues, the model gets improved and will provide a more reliable prediction. MEUZZ updates the seed selection model in three steps. First, it infers the descendant trees (5) of those seeds selected to the concolic engine in (4); then, it derives a label (6) based on the descendant trees of the previously selected seed (§4.4); finally, it updates or retrains the model (7) depending on the type of learning process (§4.5, §4.6).

4.2 System Requirements

MEUZZ aims to predict the seed utility in a more accurate and generalizable fashion than the existing heuristic-based approaches while keeping the fuzzing efficiency intact. One of the steps that contribute the most in achieving these goals is feature extraction. MEUZZ can potentially derive various semantic features because it has access to complex program

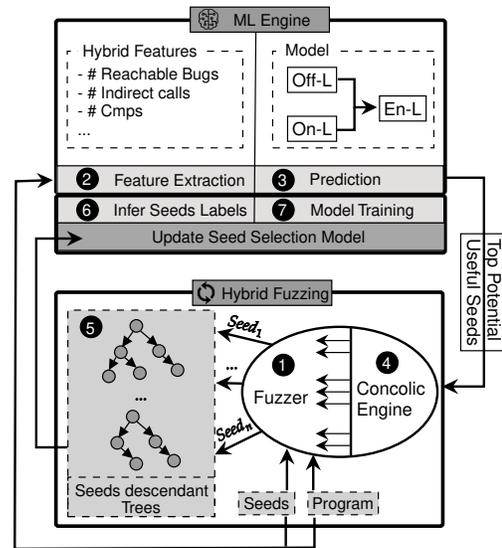


Figure 3: System overview of MEUZZ. The coordinator is extended with a ML engine, which consists of 4 modules – Feature extraction, label inference, prediction and training modules. During fuzzing, utility prediction and model training are carried out consecutively. After extracting features for inputs in the fuzzer’s queue, the ML engine can predict their utilities based on the current model. Then, with the seed labels inferred from previously selected seeds, the model is trained iteratively with the new data.

structures, such as the Control Flow Graph (CFG) with sanitizer instrumentations. However, there are some challenges that MEUZZ may encounter during feature extraction because it requires to adapt the ML engine to the online-style fuzzing workflow. To cope with such challenges, the feature engineering stage should meet the following requirements (R1–R3).

R1 - Utility Relevant: The ultimate goal of fuzzing is higher code coverage as well as discovering a higher number of hidden bugs. The features should reflect the characteristics that may improve such measures. For instance, how much a seed is likely to trigger more potential bugs or how much unexplored code a mutated seed will reach during its execution. As it is obvious, a seed is only meaningful in the context, which is the program it is executed upon. Accordingly, feature extraction needs to consider the seed and the program as a bundle.

R2 - Seed-/Program-Agnostic: To achieve generalizability, the features should be seed-/program-agnostic. If a feature is target-dependent, it downgrades the ability to generalize. For example, one could engineer a boolean feature based on the magic number that shows if a generated seed is genuine or not. Although this feature looks useful to ignore invalid seeds for fuzzing a specific program, it needs to be customized for fuzzing different programs as the inputs’ formats change. Contrarily, “meta properties” like the execution path triggered by the input are more preferable, as it is a universally usable characteristic regardless of the program.

R3 - Online Friendly: To keep the efficiency comparable to heuristic-based approaches, it is not only important how fast each feature can be extracted, but also the number of features is concerned during model construction. If the features are both light-weight and effective, it is assured that the coordinator will not be blocked from launching the concolic executor and at the same time able to construct meaningful models to predict the seed utility. As a result, suitable features should strike a balance between analysis richness (*i.e.*, how informative is the analysis result) and computation complexity (*i.e.*, what is the time complexity for the analysis).

4.3 Feature Engineering

The aforementioned requirements (R1–R3) guide us to engineer the following list of features. We discuss them in four categories.

Bug-triggering: Inspired by existing research [25], we use the number of reachable sanitizer instrumentations as guidance for measuring how likely bugs can be triggered. As sanitizer instrumentations are based on sound analysis (*i.e.*, no missed bugs), it provides a good over-approximation when trying to quantify the number of bugs that can be found. Hence, we extract these two features:

1. *Count of reachable sanitizer instrumentations:* For all branches throughout the path triggered by a given seed, the number of reachable sanitizer instrumentations is computed and then sum up. For instance, there are two branches in the left example of Figure 4. There are six potential bugs by following the branches, so the value for this feature is six.
2. *Count of reached sanitizer instrumentations:* For all branches throughout the path triggered by a given seed, we sum up the number of reached sanitizer instrumentations by the fuzzer. The major difference between this feature and the prior one is that this feature reflects the expectation of *immediately* solvable sanitizer bugs, while the former feature is an *indirect* reflection. For instance, the value of this feature in the right example of Figure 4 is two because the potential bugs can be directly reached by negating the constraints from b_1 and b_2 .

Coverage: Concolic execution is good at solving complex branch conditions. Hence if there are a lot of previously unsolved branches the concolic executor may encounter when executing on the given input, it will significantly improve the code coverage. The most common situations where concolic execution can help is when a conditional statement (*i.e.*, if-then-else or switch-case) exists. As the given input will only follow one of the branches, we call those branches stemmed from the same conditional statement *neighbor branches*. So we extract the following feature to estimate each seed’s potential of new coverage.

1. *Count of undiscovered neighbor branches:* For all branches along the path triggered by the given seed,

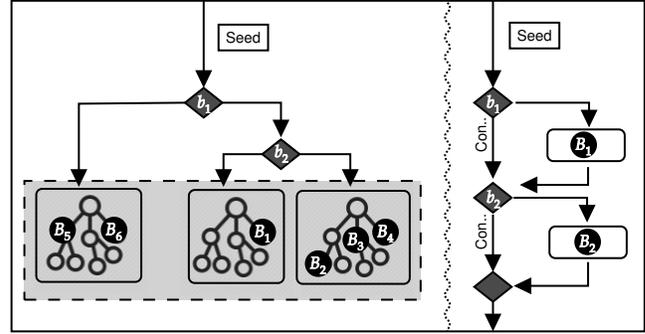


Figure 4: The examples that show how bug-triggering and coverage features are computed.

we compare their neighbors, if any, with all previously triggered branches. We then sum up the previously undiscovered neighbors for each branch. For instance, the value of this feature in the right example of Figure 4 is two if the seed follows the path with continue labels.

Constraint Solving: We also devised a set of features that impact the solving capabilities of the concolic execution engine. The incentive behind selecting such features is that the performance of the concolic executor significantly influences the entire hybrid fuzzing system.

1. *Count of external calls:* Existing concolic executors either rely on a simulated procedure or simply terminate the path execution when encountering an external function. As a result, external function calls may have negative impacts on the concolic executor, such as misleading the path and causing failure to generate correct seeds. This feature records the count of external function calls along the path executed by the given seed.
2. *Count of comparison instructions:* This feature records the count of `cmp` instructions along the path executed by the given seed. Comparison instructions pose the constraints on the execution path, which will later be solved by the SMT solver. However, constraint-solving is very time-consuming and is often the reason for the timeout.
3. *Count of indirect calls:* This is the number of indirect call instructions along the path executed by the given seed. Indirect calls may cause state explosion because when the concolic executor encounters an indirect call with a symbolic pointer, it simply forks a state for each possible value that can be resolved for the symbolic pointer [44]. In large programs, there could be many possible values for a symbolic function pointer.
4. *Length of path:* This feature records the number of executed branches (not deduplicated) by the given input. It helps identify the existence of large loops, which is another common reason that causes state explosion and solver timeout.

Empirical: This set of features is devised based on the empirical observations by existing works. They might

indirectly affect fuzzing performance.

1. *Input size*: Size of the input is often employed by existing tools as a heuristic to make a scheduling decision. On the one hand, smaller size inputs often end the execution more quickly and then leave more time for the fuzzer or concolic executor to explore other inputs [13, 53]. On the other hand, larger input has a better chance to trigger more functionalities [25]. Therefore, we consider the input size as one of the potential features for our approach.
2. *First seed with new coverage*: This is a boolean value indicating whether the given seed is the first one to discover some new branches or not. This is based on the intuition that such seeds are more likely to trigger more new coverage. This feature is used in many popular fuzzers [8, 33].
3. *Queue size*: This feature records how many inputs are saved in the fuzzing queue at the time of the query. If the queue is long, it is less likely to see more new coverage. Since MEUZZ needs to predict the utility of each seed during runtime, namely how much *more* new coverage can be discovered by fuzzing with the given input, the prediction should consider the current status of fuzzing.

4.4 Seed Label Inference

Labeling is an indispensable stage of data preprocessing in supervised learning. Well-defined labels make the prediction much easier and more reliable. As we aim to predict the utility of a selected seed and there is no direct indication to show if the selected seed is definitely useful, we need to derive a label by which we show the proportion of the seed utility.

To understand the utility of a seed, we need to fuzz the program with that seed and check the outcome. Fuzzers that use genetic algorithms (GAs) for seed generation represent such an outcome as a forest of *input descendant tree*, which depicts the parent-child relationship of the seeds in the fuzzer's queue. Each node of the tree represents a seed, and each edge connects a seed to one of its mutants.

In plain fuzzing, the root nodes are the original seeds provided by the user. Similarly, in hybrid testing, we model the inputs that are selected to be executed concolically as the root nodes. When an input is selected to explore, the concolic engine will produce mutants of the running input. These mutants can further cover the neighbor branches (§ 4.3) of the re-visited path. After these mutants get transferred back to the fuzzer's queue, the fuzzer can use GA to further mutate them. As a result, we can draw the parent-child edges from the selected input to the mutants generated by the concolic engine, and to their GA-derived offsprings to form a mega descendant tree.

If the descendant tree of a seed is larger, it comparatively means the seed contributes more to the fuzzer's code coverage. Hence, to derive the label, we measure the size of the input descendant tree of a seed and consider it as the label.

In reality, it is not feasible to compute the complete descendant tree since it could grow indefinitely if the user never terminates the fuzzing process. As a result, we have

to limit the tree analysis to a time window to make the label inference possible. Specifically, after the fuzzer imports a seed from concolic executor, we wait for a certain number of fuzzing epochs for the fuzzer to explore the imported seed and then compute the size of its descendant tree.

4.5 Model Construction and Prediction

The next step after preparing the data is to predict the seed prominence (*i.e.*, label). As the seed labels are the number of nodes in the *seed descendant tree*, their values are continuous so we need a regression model to predict them. Hence, we embed a regression model in MEUZZ in a way that when new seeds are generated by the fuzzer, the model predicts the utility of the seeds and then transfer the potential seeds to the concolic engine.

MEUZZ predicts very naively or just random at the beginning of fuzzing because the model just sees a few samples. However, the prediction becomes more reliable when more seeds are generated—data plays a crucial role in advancing model—and the model receives updates.

As seeds are mutated continuously during fuzzing a program in real-time, prediction and model update need to be done in a limited time window. Such limitation makes online learning approaches desirable candidates for model construction. In online-learning, the model can be incrementally updated by only considering new data. It does not need to store all previous data and to learn a model from scratch in every iteration. Instead, the model can be updated incrementally based on the incoming input, previous model and historical fuzzing yields. Such an update is very fast and requires less storage, which fits our use case very well. Thus we adopt online learning as one of the techniques for model construction.

4.6 Updating Model

To assure the model is entirely up-to-date with the prevailing seeds, ideally, we need to dynamically update/retrain the model, depending on the learning type (*i.e.*, online vs. offline). By doing so, we can both predict and learn in real-time.

For online learning, we use the Recursive Least Square (RLS) algorithm [21, 46] to update our linear model. Suppose at time t , the input data and the label are \mathbf{x}_t and y_t correspondingly, where \mathbf{x}_t is a vector of d -dimension. The following formula shows how the weight of the model at time t (*i.e.*, \mathbf{w}_t) is updated based on the weight obtained from the previous model (*i.e.*, \mathbf{w}_{t-1}):

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \mathbf{C}_t^{-1} \mathbf{x}_t [y_t - \mathbf{x}_t^T \mathbf{w}_{t-1}]$$

where \mathbf{C}_t^{-1} is the inverse of for \mathbf{C}_t , and \mathbf{C}_t is defined as:

$$\mathbf{C}_t = \sum_{i=1}^t \mathbf{x}_i \mathbf{x}_i^T + \lambda \mathbf{I}$$

Note that to calculate \mathbf{C}_t^{-1} , we do not need to store all previous data and compute the inverse. Based on the Woodbury formula, \mathbf{C}_t^{-1} can also be updated recursively as follows:

$$\mathbf{C}_t^{-1} = \mathbf{C}_{t-1}^{-1} - \frac{\mathbf{C}_{t-1}^{-1} \mathbf{x}_t \mathbf{x}_t^T \mathbf{C}_{t-1}^{-1}}{1 + \mathbf{x}_t^T \mathbf{C}_{t-1}^{-1} \mathbf{x}_t}$$

The complexity for such an update is $O(d^2)$.

To update the offline learning algorithms, the model needs to be retrained with all historical data in every iteration. Although retraining the model with the whole dataset every time a new seed is coming seems to be time-consuming, we show in our evaluation the approach is still practical in our case (§6.3). One reason is that the seed attributes are not of very high dimension and the number of seeds that need to be retrained is within an acceptable order of magnitude.

5 Implementation

Among the three components of MEUZZ, two of them are based on off-the-shelf software. We employ AFL-2.52b [33] for the fuzzing module and the re-engineered variant of KLEE from SAVIOR [25] for concolic execution. We develop the coordinator component from scratch in Python in 3,152 SLOC. Below, we detail the implementation of the major components of the ML engine, namely feature extraction and label inference.

Feature extraction: As discussed in §4.2, considering the trade-off between computational complexity and accuracy is key in feature extraction. Hence, for developing complicated features, we use a combination of static and dynamic analyses to offload the heavy tasks to compile time as much as possible. For instance, to extract the bug triggering features, we first instrument the target program with UBSan [15] at compile time. Then, a reachability analysis based on SVF [50] is used to extract the number of sanitizer instrumentations that can be reached from each branch. During runtime, we simply collect all the triggered branches by replaying the input and add up the number of reachable instrumentations from these branches.

To extract the feature of undiscovered neighbors, we record the branches and their neighbors at compile time. This information is later used to query whether any neighbor of a triggered branch is covered. To facilitate fast queries, we store the neighbor list as a disjoint-set data structure and use the union-find algorithm to query during runtime.

We extract the rest of the features either via compile-time instrumentation (*e.g.*, `cmp`, `call` instructions) and runtime input replay or via operating system APIs (*i.e.*, `size`, `queue size`, and `new coverage`).

Label inference: To collect the size of *seed descendant tree*, we traverse AFL’s fuzzing queue. Thanks to the seed naming system of AFL (*i.e.*, [`id`, `source`, `mutation`, `new cov`]), we can iteratively traverse the seeds and use transitive closure to collect all the inputs imported from the concolic executor and their descendant trees.

Table 1: Evaluation settings

Program			Settings	
Name	Version	Driver	Initial Seeds	Options
tcpdump	4.10.0	tcpdump	[14]	-r @@
binutils	2.32	objdump	[5]	-D @@
binutils	2.32	readelf	[5]	-A @@
libxml	2.9.9	xmllint	[11]	stdin
libtiff	4.0.10	tiff2pdf	[10]	@@
libtiff	4.0.10	tiff2ps	[10]	@@
jasper	2.0.16	jasper	[9]	-f @@ -T pnm
libjpeg	jpeg9c	djpeg	[9]	stdin

6 Evaluation and Analysis

We conduct a comprehensive set of experiments to answer the following research questions:

- *RQ1:* Can ML-based seed scheduling outperform heuristics-based approaches (§ 6.2 and § 6.6)?
- *RQ2:* Which features are more important in predicting seed utility and which learning mode is more effective (§ 6.3)?
- *RQ3:* Does the learned model adapt well to different fuzzing configurations (§ 6.4)?
- *RQ4:* Is it feasible to transfer the learned model from a program to other programs to improve fuzzing yields (§ 6.5)?

6.1 Evaluation setup

Following the general fuzzing evaluation guideline [32], we choose 8 real-world benchmark programs commonly used by existing work [19, 24, 25, 54, 55]. Table 1 shows the configurations used for fuzzing each program. All experiments are conducted on AWS c5.18xlarge servers running Ubuntu 16.04 with 72 cores and 281 GB RAM. Without explicitly mention, all tests run for 24 hours each by assigning three CPU cores to each fuzzer and are repeated at least 5 times; we report the average result with Mann-Whitney U-test.

We compare MEUZZ with the state-of-the-art grey-box fuzzers, such as AFL [33], AFLFast [19], and Angora [24], as well as hybrid testing systems including QSYM [54] and SAVIOR [25]. The seed selection modules of all these previous systems are based on heuristics. We could not test Driller [49] on the chosen benchmarks because its concolic execution engine fails to run them. Moreover, we test Vuzzer [41] and T-Fuzz [39] but we compare them with MEUZZ in a different way than we do with the other fuzzers. This separate comparison is because these two fuzzers do not support concurrent fuzzing. Due to the space limit, we discuss our observations and show the results of their branch coverage in Appendix C.

For MEUZZ, we consider three different configurations according to the learning process, namely MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, which refer to online learning linear model, offline learning random forest model and the arithmetic average of the first two models’ utility predictions, respectively.

Since Savior and QSYM need at least three CPU cores, we enforce this fuzzing setting to all the fuzzers to build a

fair comparison environment. We launch one master and two slaves for the grey-box fuzzers; and one master, one slave, and one concolic execution engine for the hybrid fuzzers. To reduce the randomness of OS scheduling, we pin each component of the fuzzers on the specific core. Because MEUZZ and SAVIOR instrument the testing program with UBSAN [15], we also apply this sanitizer to all other fuzzers, as enabling sanitizers is shown to improve the fuzzer’s effectiveness for finding bugs.

6.2 Learning Effectiveness

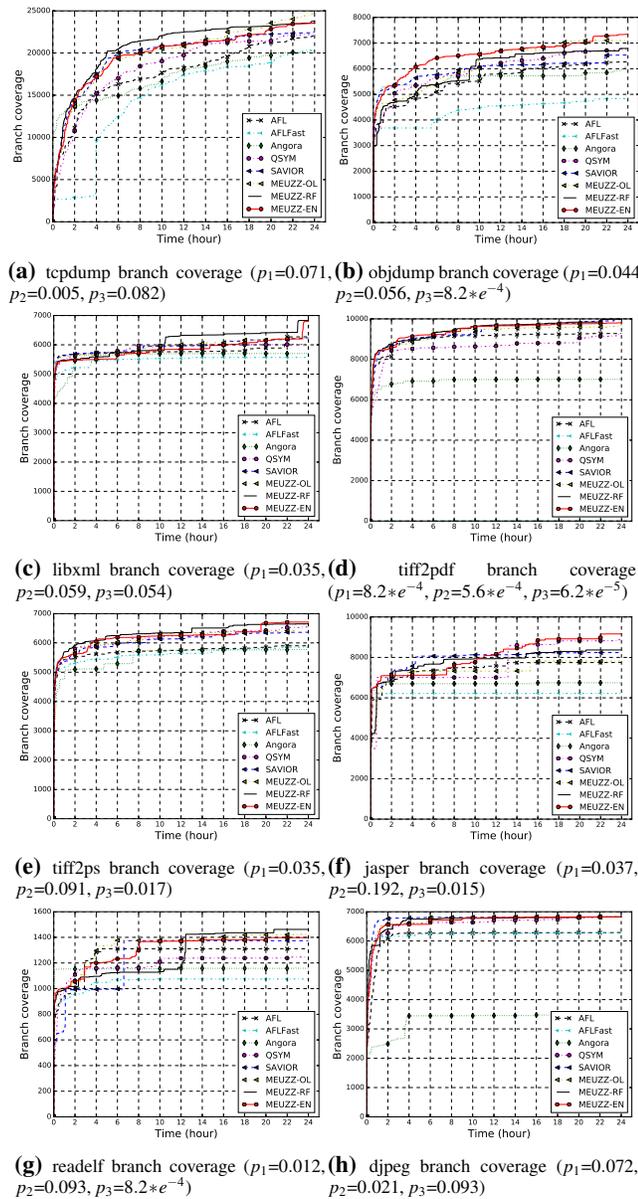


Figure 5: Branch coverage fuzzing with valid seeds (higher is better). p_1 , p_2 and p_3 are p-values in Mann-Whitney U Test by comparing QSYM with MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, respectively.

Table 2: Execution time spend on different learning stages

Model Update (s)		Prediction(s)		Feature Extraction (s)
Online	Offline	Online	Offline	
0.000636	0.326139	0.000016	0.003168	5e-6

The most straightforward metric to measure the effectiveness of MEUZZ is code coverage, which is also a widely accepted and evaluated metric. Figure 5 shows the branch coverage achieved by different fuzzers to the required time for fuzzing. Based on the coverage result, we have several interesting findings.

First, MEUZZ covers more code than other fuzzers in most programs after 24 hours of fuzzing. Among the *non-ML* fuzzers, QSYM performs the best in terms of code coverage, thanks to its efficient concolic execution engine tailored specially for hybrid fuzzing. Compared with QSYM, the MEUZZ variants achieve various levels of coverage improvements. In `tcpdump`, `objdump`, `readelf` and `libxml`, MEUZZ improves code coverage over QSYM by more than 10%, and particularly 27.1% by MEUZZ-RF in `readelf`. In `tiff2pdf` and `tiff2ps`, MEUZZ also has moderate coverage improvements. However, in `jasper` and `djpeg`, there is no much difference between MEUZZ and QSYM; we speculate it is because all fuzzers are saturated and hit a plateau after 6 hours.

Second, MEUZZ covers less code in the beginning but gradually surpasses other fuzzers as time progresses. For example, in `objdump` MEUZZ-OL and MEUZZ-RF did not cross QSYM and SAVIOR until after 9.6 hours of fuzzing, but MEUZZ eventually achieves 14% higher code coverage. Similar situations can be observed in `libxml`, `readelf` and `tiff2ps`. This observation is expected, as MEUZZ starts seed scheduling with random parameters, hence the performance of seed selection is unpredictable at the beginning. But as time passes, fuzzing data are increasingly collected and used to refine the prediction model. Hence, the prediction becomes more accurate.

Lastly, the effectiveness of ML is presented in Figure 10 in Appendix D. It is shown that different programs are variously affected by different sets of features. For instance, *External Calls* has more influence on six of the programs except for `tcpdump` and `djpeg`, showing that no single feature is sufficient to predict high-utility seeds. By using a data-driven approach, we cannot only automatically select the high impactful features in different programs or situations, but also integrate them in a more optimal way than manual-crafting rules.

6.3 Insights and Analyses

Online v.s. Offline learning: As mentioned in the previous section, offline learning with the random forest model sometimes beats online learning with the linear model; however, the main concern with using offline learning is time delays, especially during the model updating stage.

To further analyze the effects of time delays caused by offline learning, we profile each learning stage during the 24

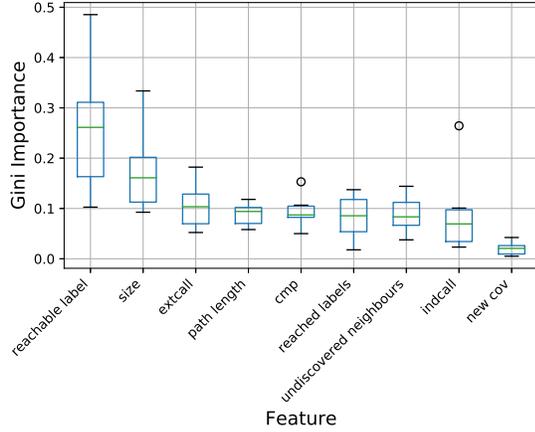


Figure 6: The box plots show the importance of the features on nine programs. The importance is extracted by training an offline random forest model and they are ranked by the median of their importance. *Reachable label* and *New Cov* are the most and the least important ones, respectively.

hours of fuzzing and report the average time spend on different learning steps. As shown in Table 2, although offline learning spent 512x and 198x more time than online learning on updating the model and making predictions, respectively, the absolute time-lapse is negligible (*i.e.*, in milliseconds). Hence, offline learning is not a critical hindrance throughout the hybrid fuzzing loop, which endorses the offline learning effectiveness discussed in Section 6.2. Having said that, if fuzzing continues for a longer time and the number of seeds significantly increases, offline learning can become an obstacle.

Feature Analysis: Figure 6 presents the distribution of the importance of each feature separately in all programs. The importance score is computed by capturing the *mean decrease impurity* from the offline random forest models [22]. The figure shows the contribution of the *New Cov* feature is the least among all the features. While it is difficult to entirely disregard the minor contribution of *New Cov*, this suggests that putting much effort to follow the seeds that bring new coverage might jeopardize the chance to explore unknown seeds. This is also known as the famous Multi-Armed Bandit (MAB) problem [18]. This finding might shed some light on the scheduling algorithm implemented in the popular fuzzers like AFL [33] that heavily rely on the *New Cov* heuristics.

Also, the variance of change in the figure shows some of the features like *Path Length* and *New Cov* are less subject to programs, while others like *Reachable Label* are more tied to programs. If the extraction of a feature heavily depends on static analysis, it is less precise compared with dynamic analysis because the sensitivity of static analysis affects the precision (*i.e.*, flow/context/field sensitivity). We speculate this is one of the reasons that make a feature (*e.g.*, *Reachable Label*) more dependent on individual programs. Also, there are additional factors that might affect dependability. Program loops as a

common trait in all programs uniformly affect the *Path Length* feature, which makes the feature more agnostic to programs. Similarly, *New Cov* is set to a seed during runtime when it is the first one to trigger new behaviors (*e.g.*, coverage); this attribute is generally applicable to a variety set of programs.

It is worth noting that the average time to extract each feature is only $5\mu s$ (as shown in Table 2), thanks to our light-weight feature extractions. This indicates that the online-friendly requirement is satisfied in MEUZZ.

6.4 Model Reusability

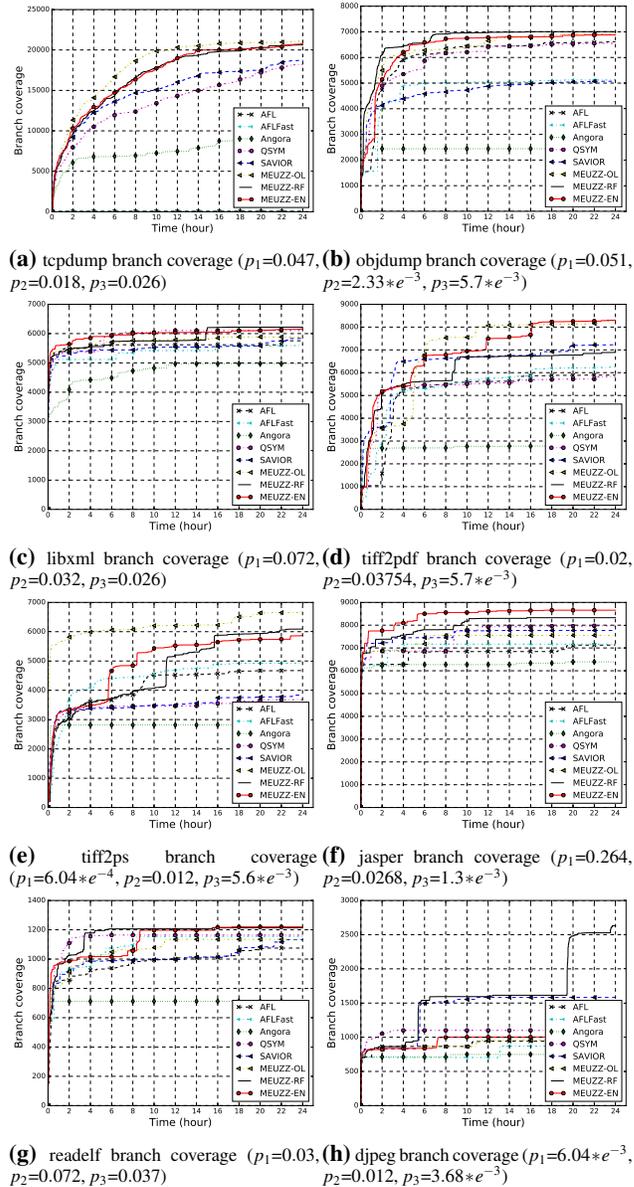


Figure 7: Branch coverage fuzzing with naive seeds (higher is better). p_1 , p_2 and p_3 are p-values in Mann-Whitney U Test by comparing QSYM with MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, respectively.

Building machine learning models is a valuable but time-consuming task. It is reasonable to build and reuse models where possible. By reusing a model, one can improve generalization, speed up training, as well as improving the model accuracy. Also, reusability can be good evidence that our model correctly captured what kind of inputs have higher utility when testing the target programs. Hence, we test the reusability of the learned models obtained via the previous fuzzing experiments.

We conduct an experiment in which we use a pre-trained model for fuzzing the same target program and compare the coverage difference. We make the following two changes in the experiment performed in § 6.2: (i) the initial seeds are replaced by a naive input that only consists 4 whitespaces; and (ii) all MEUZZ variants are initialized with the models they learned in the effectiveness test (with valid initial seeds).

Figure 7 shows the coverage result with Mann-Whitney U Test. There are several interesting observations. The most important one is that the MEUZZ variants start performing well even at the beginning of fuzzing compared with when there is no model initialization. We believe this improvement is brought by the initial models. Additionally, “pure-AFL” fuzzers do not perform well with this naive initial seed. For instance, in `tcpdump`, AFL and AFLFast only generate 6 inputs in total after 24 hours of fuzzing (see Figure 7a). On the contrary, systems augmented with other input generation techniques such as concolic execution and taint analysis can generate more inputs and consequently can explore significantly more code. Lastly, MEUZZ-RF outperforms its peers in `djpeg`, and its p-value indicates the improvement is significant (< 0.05), suggesting the non-linear model works better on `djpeg`.

6.5 Model Transferability

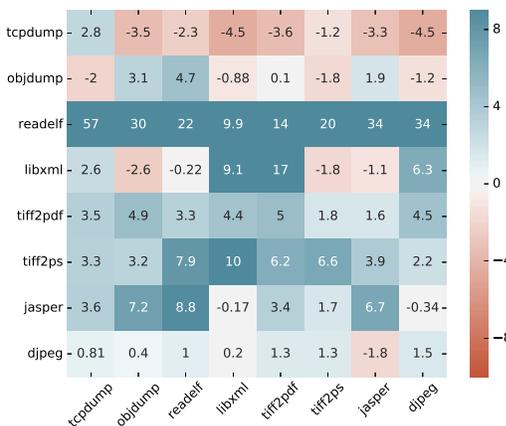


Figure 8: This heat map shows Coverage improvement with model initialization for MEUZZ-OL over vanilla MEUZZ-OL. Y-axis is the tested programs, X-axis is the models used for initialization. Each cell shows the relative coverage comparison (%). The diagonal values show the coverage improvement on each program after initializing MEUZZ with model learn from the same program (reusability). Model transferability is shown in 7 out of the 8 programs.

To further evaluate the model reusability explained in the previous section, we conduct a cross-program experiment to determine whether a model trained on one type of program will transfer well to fuzzing a new program. This is known as transfer learning in the ML field [38]. As far as we know, no prior research has attempted to show this invaluable analysis in fuzzing [42].

In this experiment, we augment MEUZZ with a pre-trained model from one program and compare the result of the fuzzer on different programs with a baseline. Our baseline is the coverage result from the learning effectiveness experiment (§ 6.2), in which we use valid seeds to bootstrap fuzzing without model initialization. We choose MEUZZ-OL as the representative of our system to measure this transferability experiment. We then fuzz each program using MEUZZ-OL initialized with the 8 pre-learned models; the models are fixed afterward.

Figure 8 visualizes the comparative coverage improvements (*i.e.*, percentage) produced by each fuzzing configuration. The Y-axis shows the tested program and the X-axis shows the programs by which the models are built. This result shows three interesting findings.

First, MEUZZ-OL observes 7.1% more code coverage on average when it is tested on the same program it is initialized with. The amount of improvement for each program is shown in the diagonal of Figure 8, from top left to bottom right. Note that these models are only learned in 24 hours from previous experiments; we expect to see more improvement in continuous fuzzing services (*e.g.*, [12]). This again confirms that the previously learned models are reusable.

Second, MEUZZ-OL observes improvement in 38 out of 56 cross-testing cases, which shows 67.9% success rate when the model is transferred from a program to another program. Among them, 10 cases see more than 10% coverage improvement. Such improvement also indicates that the program-agnostic requirement is satisfied in MEUZZ.

Last but not least, we notice different programs have different “sensitivity” towards the transferred models. For instance, almost all the transferred models can strengthen fuzzing `readelf`, `tiff2pdf`, `tiff2ps` and `djpeg` programs, among which `readelf` sees the highest improvement. Interestingly, `readelf` achieves even higher improvement when using the `tcpdump` model than the `readelf` model by itself. However, other programs are only partially accepting foreign models. For instance, the model of `tcpdump` can outperform almost all of the programs, while none of the other seven external models can improve its fuzzing yields.

Two main reasons can justify the aforementioned observation, namely the number of data points as well as feature importance. When there is more data, the model can better generalize [31]. For instance, the `tcpdump` model contains a higher number of seeds compared with others (see Figure 1), which justifies the effectiveness of the transferred model built from the `tcpdump` program. We also compared the importance of the features of each program (see Appendix D). The shape

Table 3: The table shows the unique bugs found by all evaluated fuzzers.

Program	AFL	AFLFast	Angora	QSYM	Savior	MEUZZ	All unique bugs
tcpdump	14	13	12	11	12	14	14
objdump	2	2	5	2	8	6	9
readelf	3	2	5	5	4	4	6
tiff2pdf	1	1	1	2	2	2	2
tiff2ps	1	2	2	5	4	6	6
jasper	2	1	0	3	1	6	8
djpeg	9	7	7	9	9	9	9
Total	32	28	32	37	40	47	54

of the final importance chart in `tcpdump` diverges more from the rest of the programs. Moreover, the values of some features such as *Indirect Call* and *Path Length* are higher than other programs. By looking at these statistics as well as checking the source code of `tcpdump` we noticed `tcpdump` is designed with heavier use of function handlers for different types of network packets and recursive loops for parsing packet fields. While other models contain different feature value distribution as well as fewer data points, which justify the failure of using them to improve fuzzing `tcpdump`.

6.6 Discovered Bugs

To prove the effectiveness of our system in discovering new bugs, we performed various analyses. We manually analyzed all of the reported undefined behaviors and crashes. UBSan reports a large amount of undefined behaviors; however, the majority of them are deemed benign after our triage process. We also triage additional bugs with the help of ASAN [1] and LeakSAN [7].

Table 3 shows our triage result for all the fuzzers. In total, 54 unique bugs were uncovered. MEUZZ outperforms other fuzzers and found 47 unique bugs, which supports the fact that higher code coverage correlates to a higher number of triaged bugs. Due to space limit, we present more detailed triage result and one of the discovered bugs only found by MEUZZ in Appendix B. This result shows MEUZZ is more effective in terms of finding bugs than state-of-the-art systems with manually crafting heuristics.

7 Related Work

7.1 ML for Fuzzing

Despite the promising potential to improve fuzzing, the application of ML has not been very-well investigated in the past and only a few research have leveraged ML. ML can be integrated into various stages of fuzzing, from input generation to crash categorization.

Input generation: The most intelligent stage of fuzzing has been the input generation stage so far, thanks to genetic algorithms. Deep learning (DL) techniques have been recently applied to input generation for both mutation/generation-based fuzzing. Such approaches [30, 40, 47] use various neural network methods to learn the patterns that exist in input files and then identify the likely input forms to trigger new

coverage. Similarly, reinforcement learning (RL) [20] can learn input grammar for generation-based fuzzers.

Crash analysis: Automating the analysis of outputs/crashes generated by fuzzers is another ML application. For instance, ML can be used to categorize crashes by identifying the root cause of them. This helps remove duplicate outputs and therefore reduces manual analysis effort [26]. Or another example is employing ML to predict whether the reported crashes by fuzzers are exploitable [52].

To the best of our knowledge, there has not been any research that practices ML for seed selection. In general, the practicality of ML for fuzzing has not been shown clearly in the past due to the uncertainty about reusability and transferability.

7.2 Seed Scheduling Heuristics

Scheduling in fuzzing: FuzzSim [51] models the seed scheduling problem as a weighted coupon collector problem and found out that scheduling can have a direct impact on fuzzing campaign yields. Later, in grey-box fuzzing, AFL [33] implements a scheduling algorithm that consists of simple heuristics such as preferring first seed with new coverage, and with smaller size and less execution time. This simple algorithm is later improved by Fairfuzz [34] and AFLFast [19] which steer the fuzzer towards less explored paths.

Scheduling in hybrid testing: As hybrid testing becomes more popular, seed scheduling also becomes a research topic. Driller [49] implements a random scheduling algorithm, while QSYM [13] implements heuristics similar to AFL. Later, DigFuzz [55] shows the ineffectiveness of random scheduling and proposes a Monte-Carlo model to predict the difficulty of each path explored by the fuzzer by far, and send the most difficult ones to concolic executor. SAVIOR [25], on the other hand, uses bug-driven scheduling heuristics. By selecting the seeds that can reach more sanitizer instrumentations, it triggers more bugs in the given timeframe than other fuzzers.

Compared with these approaches, MEUZZ applies machine learning techniques that can learn a utility prediction model, which is adaptive to the program being tested. As our evaluation suggests, this approach is more scalable and more performant than the manual-crafting scheduling heuristics.

8 Conclusion

We present MEUZZ, a hybrid fuzzing system featuring machine learning and data-driven seed scheduling. Theoretically, MEUZZ is more generalized than systems using fixed seed selection heuristics. For effective integration of machine learning workloads into the online hybrid fuzzing loop, MEUZZ follows the requirements of being utility relevant, online friendly and program agnostic for its feature engineering and label inference. Our evaluation shows that MEUZZ outperforms state-of-the-art fuzzers in both code coverage and bug discovery. In addition, the learned models demonstrate good reusability and transferability, making it more practical to apply machine learning to hybrid fuzzing.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This project was supported by the National Science Foundation (Grant#: CNS-1748334) and the Office of Naval Research (Grant#: N00014-17-1-2891). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- [1] Addresssanitizer. <https://clang.llvm.org/docs/AddressSanitizer.html>.
- [2] Afl technical details. http://lcamtuf.coredump.cx/afl/technical_details.txt.
- [3] angr/tracer: Utilities for generating dynamic traces. <https://github.com/angr/tracer>.
- [4] Announcing oss-fuzz: Continuous fuzzing for open source software. <https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html>.
- [5] Binutils test cases. <https://github.com/mirrorer/afl/tree/master/testcases/others/elf>.
- [6] Darpa cyber grand challenge. <http://archive.darpa.mil/cybergrandchallenge/>.
- [7] Leaksanitizer. <https://clang.llvm.org/docs/LeakSanitizer.html>.
- [8] libfuzzer – a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>.
- [9] Libjpeg test cases. <https://github.com/mirrorer/afl/tree/master/testcases/images/jpeg>.
- [10] Libtiff test cases. <https://github.com/mirrorer/afl/tree/master/testcases/images/tiff>.
- [11] Libxml test cases. <https://github.com/mirrorer/afl/tree/master/testcases/others/xml>.
- [12] Oss-fuzz - continuous fuzzing for open source software. <https://github.com/google/oss-fuzz>.
- [13] Qsym: A practical concolic execution engine tailored for hybrid fuzzing. <https://github.com/sslabs-gatech/qsym>.
- [14] Tcpdump test cases. <https://github.com/the-tcpdump-group/tcpdump/tree/master/tests>.
- [15] Undefined behavior sanitizer - clang 9 documentation. <http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks>.
- [16] When does deep learning work better than svms or random forests? <https://www.kdnuggets.com/2016/04/deep-learning-vs-svm-random-forest.html>, 04 2016.
- [17] Cyber grand shellphish. http://www.phrack.org/papers/cyber_grand_shellphish.html, 2017.
- [18] Donald A Berry and Bert Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5:71–87, 1985.
- [19] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.
- [20] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. *CoRR*, abs/1801.04589, 2018.
- [21] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142.
- [22] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.
- [23] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
- [24] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
- [25] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 2–2, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.
- [26] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1084–1093, June 2012.

- [27] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.
- [28] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, 15(1):3133–3181, January 2014.
- [29] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [30] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.
- [31] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, March 2009.
- [32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.
- [33] lcamtuf. american fuzzy lop. <http://lcamtuf.coredump.cx/afl/>, 2015.
- [34] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *CoRR*, abs/1709.07101, 2017.
- [35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Network and Distributed System Security Symposium (NDSS)*, 2018.
- [36] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.
- [37] M. Nayrolles and A. Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 153–164, May 2018.
- [38] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct 2010.
- [39] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
- [40] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR*, abs/1711.04596, 2017.
- [41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
- [42] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.
- [43] Adam Santoro, David Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, Peter W. Battaglia, and Timothy P. Lillicrap. A simple neural network module for relational reasoning. *CoRR*, abs/1706.01427, 2017.
- [44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.
- [45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, pages 28–28. USENIX Association, 2012.
- [46] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [47] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*. IEEE, 2018.
- [48] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE, 2015.
- [49] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

- [50] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.
- [51] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.
- [52] G. Yan, J. Lu, Z. Shu, and Y. Kucuk. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 164–175, Aug 2017.
- [53] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Pro-fuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE.
- [54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 745–761. USENIX Association, 2018.
- [55] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

Appendix A Why use UBSAN

Note that although the design of MEUZZ is generically compatible with mainstream sanitizers [15, 45, 48], our implementation uses UBSan for the following reasons: (i) UBSan instruments programs with pure static checks that can be easily converted to solvable SMT constraints. In contrast, other sanitizers, such as ASAN and MSAN, employ red-zones and status bitmap, which are less amenable to constraint solving. (ii) Our concolic engine is based on SAVIOR’s KLEE, which uses UBSan as the primary sanitizer. Using UBSan makes concolic execution more effective as shown in [25].

Appendix B Bugs found by MEUZZ

We provide a more detailed triage information of the bugs found by MEUZZ. In total, MEUZZ found 30 undefined behaviors, among which 21 have been confirmed/fixes so far by the developers and the rest are pending. For the reported bugs, we found the potential UBs with UBSAN [15] and manual analysis; we found the memory errors and DoS with ASAN [45] and memory leaks with LeakSAN [7].

Table 4: The table shows the discovered bugs by MEUZZ. UB, ME, DoS, and ML refers to Undefined Behavior, Memory Error, Denial of Service, and Memory Leak, respectively.

Program	Potential UB	ME	DoS	ML	Confirmed
tcpdump	14				4
objdump	4			2	
readelf	2	1		1	1
tiff2pdf			2		2
tiff2ps	1	4	1		4
jasper			4	2	4
djpeg	9				6
Total	30	5	7	5	21

```

1  for (; cc < tf_bytesperrow; cc += samplesperpixel)
2      ↪ {
3      adjust = 255 - cp[nc];
4      switch (nc) {
5      case 4: c = *cp++ + adjust; PUTHEX(c, fd);
6      case 3: c = *cp++ + adjust; PUTHEX(c, fd);
7      case 2: c = *cp++ + adjust; PUTHEX(c, fd);
8      case 1: c = *cp++ + adjust; PUTHEX(c, fd);
9      }

```

Figure 9: Off-by-one heap read overflow in tiff2ps.

One of the heap overflow vulnerabilities in tiff2ps is discovered only by MEUZZ. Figure 9 shows the vulnerable code snippet. This bug has been confirmed and fixed by the developers. It is an out-of-bound read vulnerability that can lead to information disclosure. The vulnerability takes place at PSDDataColorContig function where cp buffer with the size of 4 bytes is allocated in heap and the 5th element of the buffer is accessed by cp[4] which leads to out-of-bound read. To trigger this bug, the loop needs to be executed without early breaks. Moreover, to control the buffer size, the input needs to satisfy many constraints in the TIFFScanlineSize function so that it will return value 4. Based on the feature importance of tiff2ps (Appendix D), Size, Cmp and External Call play more important roles in its model, we believe this is why MEUZZ is able to guide the fuzzer to explore and trigger this bug. On the contrary, by replaying the fuzzing corpora, we found that other fuzzers miss this bug because they either exit the loop early or fail the checks in TIFFScanlineSize.

Appendix C Discussion on Extra Experiments

We attempted to compare MEUZZ with many state-of-the-art fuzzing systems but cannot conduct an apple-to-apple comparison with some of them due to various reasons.

Driller uses [3] as its concolic engine, which has limited support for system calls, causing the engine’s failure to generate new test cases. Similar issue was also reported by Insu at el. [54]. Vuzzer and T-Fuzz do not have support for concurrent fuzzing. After discussing with the developers we assigned only one core to them and run them for 72 hours

(3×24) instead of 24 hours. We report the branch coverage results of Vuzzer and T-Fuzz in Table 5.

Table 5: The table shows the number of branches covered by Vuzzer and T-Fuzz. **X** means fuzzer crashed on the program.

Program	Vuzzer	T-Fuzz
tcpdump	1103	11566
objdump	711	4216
readelf	1025	842
libxml2	715	X
tiff2pdf	X	4892
tiff2ps	X	3534
jasper	X	6084
djpeg	1317	763

Appendix D Detailed Feature Importance Study

Figure 10 demonstrates how the randomly initialized model evolved with more and more training data available during fuzzing. MEUZZ automatically identified which features are more important for each specific programs, showing it is more scalable than manually-written heuristics.

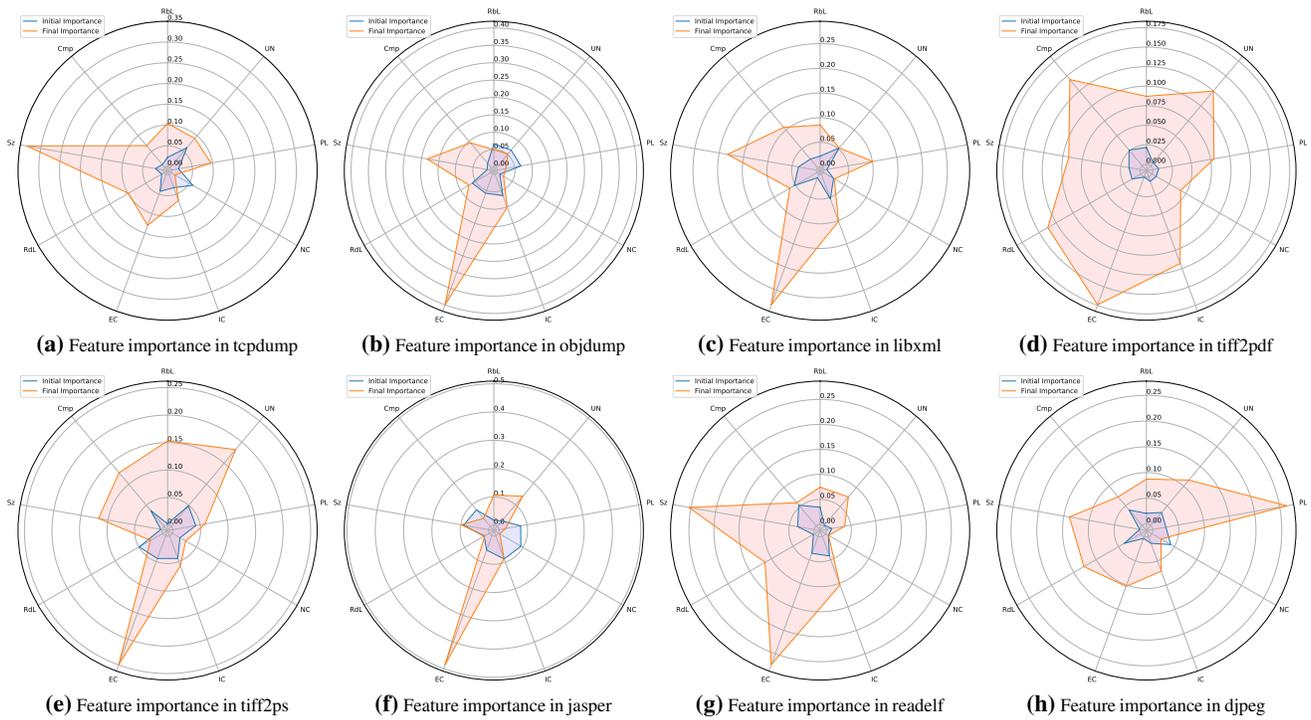


Figure 10: Feature importance extracted from models learned in the effectiveness test (§ 6.2). The initial importances are randomly generated. Sz: Size, RdL: Reached Label, EC: External Call, IC: Indirect Call, NC: New Coverage, PL: Path Length, UN: Undiscovered Neighbors, RbL: Reachable Labels, Cmp: Comparisons.

Tracing and Analyzing Web Access Paths Based on User-Side Data Collection: How Do Users Reach Malicious URLs?

Takeshi Takahashi[†], Christopher Kruegel[‡], Giovanni Vigna[‡], Katsunari Yoshioka^{*}, Daisuke Inoue[†]
[†]*National Institute of Information and Communications Technology,*
takeshi_takahashi@ieee.org, dai@nict.go.jp
[‡]*University of California, Santa Barbara, {chris, vigna}@cs.ucsb.edu*
^{*}*Yokohama National University, yoshioka@ynu.ac.jp*

Abstract

Web access exposes users to various attacks, such as malware infections and social engineering attacks. Despite ongoing efforts by security and browser vendors to protect users, some users continue to access malicious URLs. To provide better protection, we need to know how users reach such URLs. In this work, we collect web access records of users from their using our browser extension. Differing from data collection on the network, user-side data collection enables us to discern users and web browser tabs, facilitating efficient data analysis. Then, we propose a scheme to extract an entire web access path to a malicious URL, called a hazardous path, from the access records. With all the hazardous paths extracted from the access records, we analyze web access activities of users considering initial accesses on the hazardous paths, risk levels of bookmarked URLs, time required to reach malicious URLs, and the number of concurrently active browser tabs when reaching such URLs. In addition, we propose a preemptive domain filtering scheme, which identifies domains leading to malicious URLs, called hazardous domains. We demonstrate the effectiveness of the scheme by identifying hazardous domains that are not included in blacklists.

1 Introduction

The Internet has become indispensable social infrastructure, and many people access the web multiple times a day in the course of their daily lives. However, web access exposes users to diverse types of attacks, including malware infections and social engineering attacks. Various techniques have been investigated and implemented to minimize the risk of accessing malicious URLs. In particular, browsers have become more resistant to malware infections and other web-based attacks. For example, the Google Chrome browser uses Google's blacklist service, i.e., Google Safe Browsing (GSB) [1], to minimize users' access to malicious URLs. However, users still reach malicious URLs through web browsing.

To better understand how users reach malicious URLs and to improve user protection, in this paper, we introduce a per-user data analysis scheme and demonstrate its usability. First, we collect access records of each user using our browser extension. User-side data collection enables us to access data that network-side data collection cannot access, i.e., user IDs, browser tab IDs, and navigation information, which, in turn, enable us to efficiently analyze the data in detail. Then, we propose a scheme to extract an entire web access path to a malicious URL, called a hazardous path. The proposed scheme takes advantage of collected data that identify users and browser tabs to efficiently reconstruct the hazardous paths. The first accesses of the paths, i.e., entry points, are identified based on the navigation information that identifies the cause of the navigation. With the extracted access path information, we analyze the web access activities of users who reach malicious URLs, i.e., victims¹, to better understand them. First, we analyze the entry points of hazardous paths to understand the proportion of accesses via bookmarks. We then analyze the risk level of bookmarked URLs. We also analyze the time required to reach malicious URLs and the number of concurrently active browser tabs to demonstrate the usability of data collected at the user side for further analysis.

In addition, we propose a preemptive domain filtering scheme. This scheme determines the risk level of accessing each domain by analyzing the hazardous paths and identifies domains that lead to malicious URLs, i.e., hazardous domains. These domains are not blacklisted; however, the proposed scheme suggests filtering traffic on them because, even if they do not host any malicious contents themselves, the access paths thereafter lead to malicious URLs.

Contributions The primary contributions of this work are summarized as follows.

1. We describe a user-side data collection approach using a browser extension. Differing from earlier work that

¹These users are not necessarily harmed by accessing malicious URLs; however, for convenience, we use the term "victim" for such users.

collects traffic data on network devices, user-side data collection enables us to access to a broader range of data, including user IDs, browser tab IDs, and user navigation information, allowing us to efficiently analyze user behaviors.

2. We introduce a scheme that reconstructs hazardous paths from the collected data. The scheme repeatedly traces previous accesses until it reaches the entry points of hazardous paths to reconstruct the paths. The entry points are determined by looking up user navigation information. Differing from earlier work that does not discern users and browser tabs, this scheme minimizes ambiguity by discerning them, narrowing down the lines of logs that need to be analyzed.
3. We analyze users' browsing behaviors to demonstrate the usability of data obtained from browsers, i.e., tab IDs and user navigation information. The analysis reveals that bookmark access is the major type of initial access on hazardous paths. In the collected data, bookmark access occupies the largest share of entry point types on hazardous paths, and its share is greater on hazardous paths than the share on all paths, including hazardous and non-hazardous paths.
4. We analyze the risk level of bookmarked URLs by defining a parameter that indicates the certainty of reaching malicious URLs. We show that there are bookmark entries that surely lead to malicious URLs, which indicates that reviewing and sanitizing bookmark entries may minimize the risk of reaching malicious URLs.
5. We introduce a preemptive domain filtering scheme that filters traffic before users reach malicious URLs. Our analysis shows that there are non-malicious domains that often lead users to malicious URLs. To identify such domains, the scheme calculates the risk levels of all domains appeared on hazardous paths and identifies domains that are likely to navigate users to malicious URLs. The scheme may filter traffic traversing on these domains, or at least provide alerts to protect users from reaching malicious URLs.

To the best of our knowledge, this is the first academic paper that analyzes users' web access records by discerning users and browser tabs and by considering user navigation information. In particular, we focus on the entry points of hazardous paths, revealing the importance of reviewing bookmarks. The proposed preemptive domain filtering scheme is also unique, which identifies hazardous domains that are not included in blacklists.

Organization of this paper The remainder of this paper is organized as follows. Section 2 presents our data collection scheme. Section 3 introduces a scheme to reconstruct an entire hazardous path by iteratively tracing previous accesses

until reaching path entry points, and Section 4 presents an analysis of users who reach malicious URLs. A scheme to preemptively filter traffic based on domain risk levels is introduced in Section 5. Section 6 considers issues that are not addressed in the earlier sections. Related work is reviewed in Section 7, and Section 8 concludes this paper.

2 User-Side Data Collection

Web access records are collected at the user side using a browser extension developed in-house [2]. The browser extension runs on the Chrome browser, and anyone who agrees to the terms and conditions can install and use the browser extension. This browser extension works as a sensor that records each user's web access activities and periodically shares the recorded information to a server. The browser extension is in Japanese; therefore, we assume that the users are primarily Japanese speakers. In this section, the list of the collected information, the ethical considerations, the access log complementation, and the dataset generated for analysis are elaborated.

2.1 Collected Data

Our browser extension uses Google Chrome's APIs [3] to collect web access data of a user from the web browser. The browser extension uses the `chrome.webRequest` API [4] to observe and analyze traffic and the `chrome.webNavigation` API [5] to receive notifications about the status of navigation requests. The following data are collected.

1. **URL**: the URL of the requested document obtained from the HTTP request header
2. **Timestamp**: the UNIX time when the request is issued
3. **Referer**: the referer value obtained from the HTTP request header
4. **Tab ID**: the identifier of a browser tab. Its uniqueness is only guaranteed during the session.
5. **Tab URL**: URL shown on the browser tab
6. **Resource type**: the type of resource defined by the `chrome.webRequest` API. It takes one of the following values: "csp_report," "font," "image," "main_frame," "media," "object," "ping," "script," "stylesheet," "sub_frame," "websocket," or "xmlhttprequest." Note that main frame, which is identified by the value "main_frame," is a document that is loaded for a top-level frame.
7. **Transition type**: the cause of the navigation defined by the `chrome.history` API [6]. It takes one of the following values: "auto_bookmark," "auto_subframe," "form_submit," "generated," "keyword," "keyword_generated," "link," "manual_subframe," "reload," "start_page," or "typed."

In addition to the data listed above, the browser extension and server provide the following supplementary information.

1. **User ID:** unique user identifier
2. **Source tab ID:** the identifier of the tab that has generated the current tab. It is recorded when a new window is generated, for instance, by `target="_blank"` and `window.open()`.
3. **GSB evaluation results:** an indicator whether the URL is listed by GSB. The server aggregates all URLs collected in a single day and queries GSB regarding their maliciousness once a day.
4. **Alexa Traffic Rank:** traffic rankings provided by Alexa Top Site [7]. The server aggregates all URLs collected in a single day and queries Alexa Top Site regarding their ranking. Note that we check up to the top 1,000 sites.

One access record consists of the above 11 items. Note that our browser extension can collect a broader range of data; however, data not relevant to this work have been omitted.

2.2 Ethical Considerations

We worked with our Internal Review Board to ensure that our usage of the logs was ethical and respectful of users' privacy. We defined the terms and conditions for our browser extension [2]. All collected data are listed, and we stipulate that we analyze the data collected from the browser extensions to detect and prevent access to malicious URLs. Users installing the browser extension need to agree to the terms and conditions.

The collected data contains privacy-related details; therefore, we have strict restrictions on its use. Any personally identifiable user information was expunged or coded before records were stored on the servers. The user ID recorded in the log is an internal number unique to each user and cannot be directly linked to any personally identifiable information. Raw URLs cannot be shared with external parties. Therefore, we do not use VirusTotal [8], which requires us to submit raw URLs. Instead, we use GSB to evaluate the maliciousness of URLs because it does not require us to upload raw URLs. In addition, we delete all records of users who request that their records be deleted.

The logs we use in our analysis are stored on a server in a secure facility, and only registered users from registered machines that implement adequate security measures can access them. No raw data are allowed to be copied outside the machines; therefore, all analyses must be conducted on the secure servers. Only aggregated results were exported from the secure server for further analysis.

2.3 Access Log Complementation

Our scheme collects users' web access data on a granular scale, which enables us to analyze their behavior in detail.

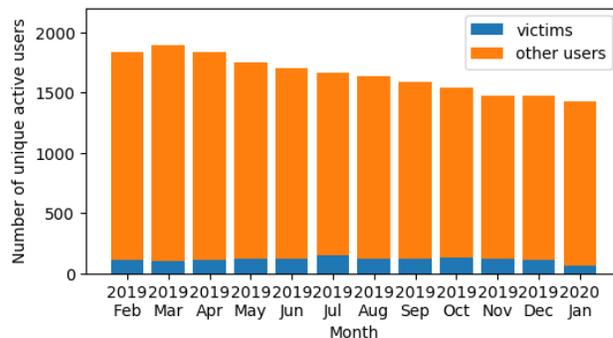


Figure 1: Number of unique active users per month

However, we sometimes fail to collect certain data. We have identified three factors influencing these failures. First, the implementation of the data collection module was imperfect. A user may take an explicit action to move to another page before the browser receives a response to a request. In this case, some fields of the access record are left blank, or the entire record is not recorded. Second, users can specifically avoid data collection for certain cases. Users can specify the list of URLs and domains that they do not want our browser extensions to record. The browser extension also does not record logs if a user accesses the web in incognito mode. These functionalities are provided to protect users' privacy. In addition, some users may deactivate the browser extension when accessing sites that they do not want us to monitor. Third, the Chrome browser does not provide the intended records when its processing burden becomes high.

Our scheme complements missing main frame entries in the log because these entries play a core role in our analysis. Typically, access to a tab URL produces a series of requests for multiple content, e.g., main frame, subframe, image, script, and style file. However, the log sometimes lacks main frame that should appear. In this case, we generate a complementary main frame entry, where its tab URL and URL are both set to the tab URL. Here the timestamp is set to that of the first entry in the consecutive access records for the tab URL.

2.4 Dataset

The dataset we used consists of data collected from February 1, 2019 to January 31, 2020 (12 months). During the experimental period, 4,306,529,287 access records were collected, among which were 76,474 accesses to malicious URLs. Figure 1 shows the number of unique active users per month. Note, a user is considered active if any browsing activity of the user is logged. On average, 831 users accessed the web at least once a day, 1,650 users accessed the web at least once a month, 1,013 users accessed the web more than seven days a month, and 115 users (victims) accessed malicious URLs at least once a month.

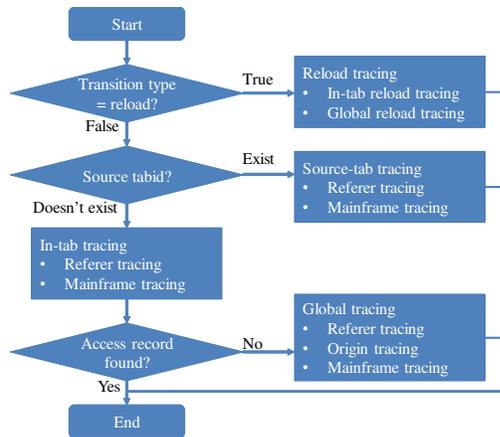


Figure 2: Process flow for tracing previous accesses

3 Access Path Reconstruction

This section introduces our access path reconstruction scheme. It extracts a hazardous path by recursively tracing previous accesses until the entry point to the path is identified. We also present case studies in this section to demonstrate the effectiveness of the scheme.

3.1 Previous Access Tracing

Our access path reconstruction scheme traces previous accesses by using user IDs and tab IDs provided by our dataset to minimize the scope of analysis. Figure 2 shows an overview of the tracing process. If the transition type of an access record is set to "reload," the access is considered as a page reload, and the reload tracing method is run to identify the access record of the reloaded page. Otherwise, if source tab ID is set, this access is considered to be originated from another tab, and the source-tab tracing method is run to identify the record of previous access on the source tab. Otherwise, the in-tab tracing method is run to identify the previous access record within the current tab. If no proper record is found, the global tracing method is run to identify the previous access record from the user's entire past access records. In most cases, our scheme analyzes a single user's access records in one browser tab, so the analysis is more efficient than the one that do not minimize the scope of the analysis. The details of the four methods mentioned above are elaborated below.

Reload tracing A "reload" transition type indicates that the page is reloaded. This includes the following cases: (1) The reload button/menu on the browser was pressed/selected, (2) the same URL as the previous access was entered in the address bar of the browser, (3) the session was reconstructed by selecting one of the recently closed tabs in the browser's history menu, and (4) a browser set to continue where its user left off on startup was started. In all cases, the tab URL and

URL of the previous access record remain the same as these of the current access record. For the first two cases, the tab ID remains the same; however, the latter two cases do not guarantee that the tab ID remains the same. If the transition type of an access record is set to "reload," our access path reconstruction scheme runs the reload tracing method, which determines the latest past access record with the same tab URL and URL values as the current access record. It first retrieves the past access records of the user within the same browser tab, and then retrieves all past access records of the user if no matching entry is found.

Source-tab tracing When an access record provides a source tab ID, our access path reconstruction scheme runs the source-tab tracing method, which analyzes the past access records of a user on the source tab to identify the previous access record. When the access record provides a referer information, the method traces the records between the current access and the latest past main frame access and determines the latest past access record whose URL matches the referer as the previous access record (referer tracing). If referer information is unavailable or no suitable record is found, the method determines the latest past main frame access record as the previous access record (main frame tracing).

In-tab tracing When an access record does not provide a source tab ID, our access path reconstruction scheme runs the in-tab tracing method, which analyzes the past access records of a user on the current tab to identify the previous access record. As with source-tab tracing method, the in-tab tracing method analyzes the records by running referer tracing and main frame tracing techniques to identify the previous access record. Note that most previous access records will be identified by this method because new tab creation is less frequent than access on the current tab.

Global tracing When no previous access was found by the in-tab tracing method, our access path reconstruction scheme runs the global tracing method, which analyzes the past access records of a user on all tabs to identify the previous access record. As with the source-tab and in-tab tracing methods, the global tracing method first runs referer tracing if referer information is available. There are cases the referer contains only its origin information. The global tracing method uses it if no suitable entry is found by referer tracing. The method determines the latest past main frame access record whose URL value includes the referer, i.e., origin, as the previous access (origin tracing). If referer information is unavailable or no suitable record is found, the method may run main frame tracing; however this does not necessarily provide a reliable trace and is not used in the analysis of this paper. Note that the origin tracing is not used by the source-tab and in-tab tracing methods because they can identify a reliable previous access

record by running the main frame tracing technique.

3.2 Identifying access path entry points

The processes described in Section 3.1 are iterated until they identify the first access of the hazardous path, i.e., the entry point. We regard an access that do not follow links and is discontinuous from the previous access as an entry point. The following types of accesses are considered as entry points.

Bookmark access A user may jump to the desired page by selecting a bookmark entry on the browser. We determine that the user came to the page by selecting a bookmark when the transition type is set to "auto_bookmark."

Session reconstruction Sessions can be reconstructed, for example, by selecting one of the recently closed tabs in the browser's history menu. We determine that the user came to the page by reconstructing a past session if the transition type was set to "reload" and no access was found on the same tab for more than a predefined amount of time. Note that we do not include the access records prior to session reconstruction in the hazardous path in this paper, but we could include such access records in the path for different analyses.

Web search A user can find various pages of interests by submitting a new search query on general web search engines. We determine that a new web search is initiated if (1) the URL is one of the top pages of major search engines², or (2) the transition type is "form_submit" and the URL is a search result page of major search engines³. Access records with a source tab ID are excluded because there must be precedent accesses in the specified tab. Note that site-specific search engines that are often available for many streaming services and pornography sites are not included.

Omnibar access The omnibar on the Chrome browser combines an address bar with the Google search box. Users can use the omnibar to initiate a web search or access their browsing history. The omnibar also suggests keywords to improve search results. The transition type "generated" identifies access that is based on the selection of choices provided by the omnibar.

²In this work, we use the following URLs to determine the top pages of major search engines: <https://www.baidu.com/>, <https://www.bing.com/>, <https://duckduckgo.com/>, <https://www.google.co.jp/>, <https://www.google.com/>, <https://www.yahoo.co.jp/>, and <https://www.yahoo.com/>.

³In this work, we determine URLs containing one of the following strings as search result pages: <https://www.baidu.com/s?>, <https://www.bing.com/search>, <https://duckduckgo.com/?q=>, <https://www.google.co.jp/>, <https://www.google.com/search?>, <https://search.yahoo.co.jp/search>, or <https://search.yahoo.com/search>.

Address typing Users often enter a new URL to initiate another browsing activity; however, as part of ongoing browsing activity, they also often modify the current URL in the address bar to browse another page, e.g., the top page of the current site. Therefore, we consider the page with the "typed" transition type as an entry point if the domain of the page differs from the previous page. Note that if the typed URL is one of the major search engines, we see it as an initiation of a new access path; however, we label the access as a web search rather than new URL typing.

Start page access When the Chrome browser starts, the page specified by the program argument or set as the default opens. If the transition type is "start_page," we consider that the browser was launched and that the page was specified in the program argument or set as the default page. Note that the transition type is set to "start_page" if a user accesses a link on an external application, e.g., an email application, because the Chrome browser is then launched with the URL of the link as its argument on the OS.

3.3 Case Studies

We extract hazardous paths by applying the techniques described in Sections 3.1 and 3.2. Two cases are described in this section to demonstrate their effectiveness.

Table 1 shows an entire hazardous path that reaches a URL labeled "SOCIAL_ENGINEERING" by GSB. First, the in-tab tracing method identified access to a subframe page in the same tab at 16:14:40 as its previous access using referer information. It then identified access to a main frame page in the same tab at 16:14:39 as its previous access using referer information. Then, the global tracing method identified access to a main frame page at 16:14:13 in another tab as its previous access using referer information. The in-tab tracing method then identified access to a main frame page at 16:13:54 in the same tab as its previous access using referer information. Now, we see that the page's transition type was set to "auto_bookmark;" thus we consider this page as the initial access of the hazardous path.

Table 2 shows an entire hazardous path that reaches a URL labeled "MALWARE" by GSB. Our in-tab tracing method identified the latest past main frame access in the same tab at 15:33:37 as its previous access. Note that the record of the main frame access was complemented using the scheme mentioned in Section 2.3. Our in-tab tracing method then identified the latest past main frame accesses in the same tab at 15:33:26, 15:32:35, and 15:32:08 as the previous accesses. The source-tab tracing method then identified the latest past main frame access on the source tab at 15:31:50 as its previous access. Similarly, the in-tab tracing method identified previous accesses by monitoring main frame and referer information. Identification of these previous accesses led to the initial access, which was accessed from a bookmark, at 15:26:57.

Table 1: Hazardous path on August 19, 2019 ("SOCIAL_ENGINEERING" label was set by GSB)

Time (JST)	Tab ID	URL(cropped)	Source tab ID	Transition type	Resource type	Tracing method
16:13:54	193	https://avgle.com/video/UNXIII	—	auto_bookmark	mframe	in-tab (referer)
16:14:13	193	https://avgle.com/search/video	—	form_submit	mframe	global (referer)
16:14:39	200	https://avgle.com/video/odaqWq	—	link	mframe	in-tab (referer)
16:14:40	200	https://olmsoneenh.info/ajWpZ.	—	auto_subframe	sub_frame	in-tab (referer)
16:14:41	200	https://10-81.s.cdn15.com/cr/3	—	—	image	—

mframe: main_frame

Table 2: Hazardous path extracted on August 28, 2019 ("MALWARE" label was set by GSB)

Time (JST)	Tab ID	URL(cropped)	Source tab ID	Transition type	Resource type	Tracing method
15:26:57	182	http://javtorrent.re/category/	—	auto_bookmark	mframe	in-tab (referer)
15:27:14	182	http://javtorrent.re/?s=080819	—	form_submit	mframe	in-tab (referer)
15:27:26	182	http://javtorrent.re/uncensore	—	link	mframe	in-tab (mframe)
15:28:28	182	http://javtorrent.re/?s=HEYZO-	—	form_submit	mframe	in-tab (mframe)
(omitted 18 access records)						
15:31:50	182	http://javtorrent.re/uncensore	—	link	mframe	source-tab (mframe)
15:32:08	403	https://www.google.com/search?	182	link	mframe	in-tab (mframe)
15:32:35	403	https://7mmtv.tv/zh/uncensored	—	link	mframe	in-tab (mframe)
15:33:26	403	https://www.google.com/search?	—	link	mframe	in-tab (mframe)
15:33:37	403	http://javhuge.com/Momoki%20	—	—	complemented	in-tab (mframe)
15:33:37	403	http://javhuge.com/zb_users/th	—	—	stylesheet	—

mframe: main_frame, o_referer: origin-only referer

4 Unveiling User Behavior

Web access data collected at the user side reveals user IDs and browser tab IDs, enabling us to efficiently reconstruct access paths, as described in Section 3. It also provides data that are not collected on the network, such as transition type. With the transition type information, we can identify access path entry points as described in Section 3.2, and such information can be used to analyze user behavior. In this section, we demonstrate the usability of the data collected on the user side by analyzing them and unveiling user behavior. Specifically, we answer the following questions: (1) what are the initial accesses of hazardous paths? (2) what is the risk level of bookmarked URLs? (3) how long does it take for users to reach malicious URLs? and (4) how many active browser tabs do users open when accessing malicious URLs? Answers to these questions may not directly provide any solution for improving user protections; however, they will deepen understanding of victims and will be a basis for future studies.

4.1 Path entry point analysis

Considering the measures implemented by recent browsers, accidentally reaching a malicious URL from the ISP's portal site or via a major search engine has become increasingly infrequent; however, users still reach malicious URLs through

Table 3: Types of path entries (February 2019–January 2020)

Types of initial accesses	Entries of hazardous paths	Entries of all paths
Bookmark access	3,213 (48.7%)	1,331,170 (38.6%)
Web search	955 (14.5%)	541,046 (15.7%)
Session reconstruction	814 (12.3%)	600,366 (17.4%)
Omnibar access	689 (10.4%)	581,223 (16.8%)
Address typing	646 (9.8%)	89,966 (2.6%)
Start page access	179 (2.7%)	307,536 (8.9%)
Link access	107 (1.6%)	—
Total	6,496 (100%)	4,403,471 (100%)

web browsing. To understand their browsing activities, we analyze the types of initial accesses on hazardous paths, i.e., hazardous path entry points. Table 3 shows the breakdown of the types of initial accesses on hazardous paths over 12 months. These types are identified based on the definition provided in Section 3.2. For reference, it also shows the breakdown of the types of initial accesses on all paths including hazardous and non-hazardous paths.

"Bookmark access" was the most frequent entry point. Many users reach malicious URLs via a bookmark, which shortens the path from a portal site or search engine to malicious sites. "Web search" was the second most frequent

entry point. Users enter a hazardous path by submitting a query to a general search engine. One typical scenario of this entry type was that users obtained a keyword from the web before they submitted a query to a search engine. For example, while browsing the adult section of a legitimate shopping site, a user found the identifier of a pornographic video, e.g., product code. The identifier was used to retrieve the web and, consequently, the user reached an illegitimate site, which eventually lead to a malicious URL. "Session reconstruction" accounted for third most frequent entry points. If we combine the paths before and after the session reconstructions, the path will become lengthy. Naturally, a lengthy path has a higher probability of reaching malicious URLs because more sites are visited, including those that are many hops away from the search engine result pages or browsers' start pages. "Omnibar access" was the fourth most frequent entry point, followed by "address typing." All of these path entry types shorten the path between the portal site or search engine and the malicious site. In this context, they are shortcuts in the access path to a malicious URL. "Start page access" was also found to be an entry point to hazardous paths; however, the number of this type of entry point is small compared with the other types.

Apart from these, several entry points are labeled as "link access." These are cases where the transition type of the access record is "link," though no appropriate previous access is found within the log we analyzed, making the access record discontinuous. Considering the meaning of the label "link" assigned by the Google API, there should be some previous accesses prior to the entry. The lack of previous accesses may occur due to either of the following reasons. First, previous access was performed prior to the first log record. Accesses prior to the first day of the month could not be traced because we conduct per-month analysis in this work. Second, the collected logs are incomplete, be it intentionally or accidentally, as we discussed in Section 2.3.

Figure 3 shows the breakdown of the types of initial accesses on hazardous paths for each month. The general trend of the breakdown is the same during these months except July 2019. There were several users who typed many URLs in the address bar of the browser in July 2019 and reached malicious URLs, though their motivations were unknown.

4.2 Bookmarked URL analysis

Table 4 shows the domains of bookmarked URLs, i.e., bookmark domains, that most frequently reach malicious URLs in descending order of the number of accesses. Note that only the accesses through the selection of bookmark entries are counted. As can be seen, the list includes pornography sites, illegal book/manga sharing sites, and file sharing sites. However, it also contains legitimate search engines, such as google.com and yahoo.co.jp. Considering the total number of accesses on any path, the percentage of

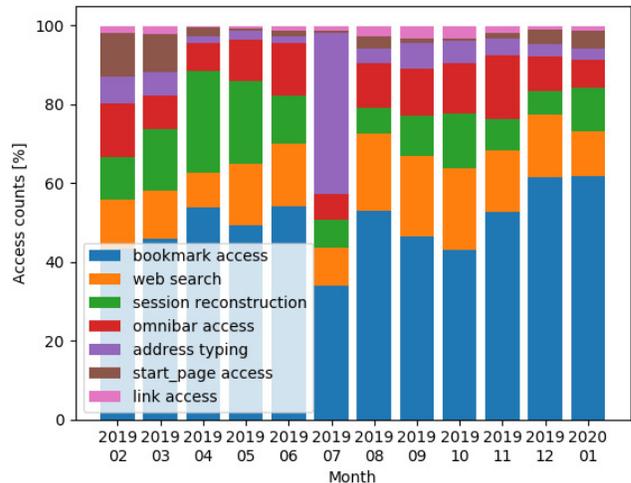


Figure 3: Breakdown of the types of path entries

Table 4: Top 10 bookmark domains that most frequently reach malicious URLs (February 2019–January 2020)

Bookmarked domains	Access counts on hazardous paths	Access counts on all paths	Risk level
avgle.com	2,184	3,566	61.25
xbooks.to	136	1,260	10.79
google.co.jp	113	69,878	0.16
yahoo.co.jp	89	191,297	0.05
bejav.net	79	79	100.00
javmost.com	37	37	100.00
mac-torrent-download.net	34	68	50.00
13dl.net	33	3,389	0.97
smv.to	19	251	7.57
youtube.com	18	95,572	0.02

accesses that reach malicious URLs remains very small; thus the risk level is small. Therefore, the risk level of domains cannot be evaluated by the number of accesses that reach malicious URLs.

To identify untrustworthy bookmark domains, we define the risk level of a domain that shows the certainty of reaching malicious URLs as follows:

$$R(domain) = \frac{NbrAccess_{malurl}(domain)}{NbrAccess_{all}(domain)} \quad (1)$$

where $R(domain)$ represents the risk level of a domain, $NbrAccess_{malurl}(domain)$ is the number of accesses to the domain that eventually reach malicious URLs, and $NbrAccess_{all}(domain)$ is the number of all accesses to the domain. Note that these parameters ignore whether a malicious URL is in the next hop or in multiple hops away. With this risk level, we evaluate the risk level of bookmark domains.

Table 5 shows the bookmark domains that most frequently

Table 5: Top 10 untrustworthy bookmark domains (February 2019–January 2020)

Bookmarked domains	Access counts on		Risk level
	hazardous paths	all paths	
bejav.net	79	79	100.00
javmost.com	37	37	100.00
xdytt.com	8	8	100.00
91mjw.com	6	6	100.00
incestflix.com	6	6	100.00
theyoump3.com	6	6	100.00
gofucker.com	5	5	100.00
anipo.tv	3	3	100.00
javbraze.com	3	3	100.00
avdvd.tv	2	2	100.00

Table 6: Top 10 Bookmark domains (February 2019–January 2020)

Bookmarked domains	Access counts on		Risk level
	hazardous paths	all paths	
yahoo.co.jp	89	191,297	0.05
youtube.com	18	95,572	0.02
google.co.jp	113	69,878	0.16
amazon.co.jp	8	45,326	0.02
google.com	6	33,494	0.02
twitter.com	3	29,159	0.01
facebook.com	2	27,387	0.01
nicovideo.jp	5	25,216	0.02
livedoor.jp	1	20,829	0.00
rakuten.co.jp	1	17,692	0.01

reach malicious URLs between February 2019 and January 2020 in descending order of the risk level. As can be seen, legitimate sites that appeared in Table 4 do not appear in the table. The top 10 bookmark domains accessed between February 2019 and January 2020 are listed in Table 6. As can be seen, the risk levels of the domains in Table 5 are significantly higher than the risk levels in Table 6.

Based on these analyses, we could review bookmarks to minimize the risk of users. As discussed in Section 4.1, bookmark access is a major entry point to hazardous paths. Therefore, we can expect to minimize the number of accesses to malicious URLs by sanitizing bookmarks. It can be difficult to prevent users from adding hazardous URLs to bookmarks or to block access to hazardous bookmarked URLs. However, simply showing alerts when accessing hazardous bookmarks may help to improve the situation. Regularly reviewing the list of bookmarks and providing alerts will be effective.

4.3 Time to reach malicious URLs

Figure 4 shows the cumulative histogram of the time to reach a malicious URL from users' first access of the paths. Among all the hazardous paths between February 2019 and January

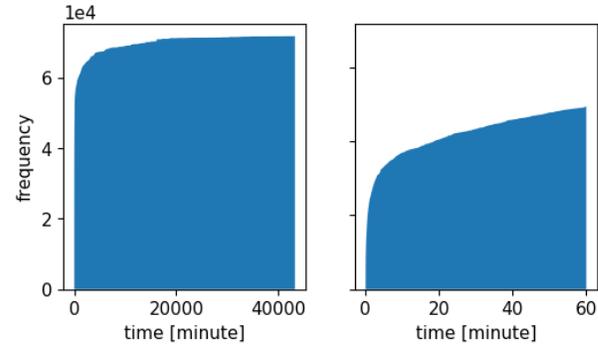


Figure 4: Normalized cumulative histograms of the time to reach a malicious URL in February 2019–January 2020 (left: complete histogram, right: first hour excerpt of the histogram)

2020, 87.03% reached a malicious URL within 24 hours from their initial accesses, 80.03% within 6 hours, 68.75% within an hour, and 60.31% within half an hour. We conjecture that many accesses to malicious URLs occur within an hour, or even within half an hour, because victims already have the URLs that are close to malicious URLs in their bookmarks, or they already know hazardous keywords that may lead to malicious URLs when initiating a web search, as seen in Section 4.1.

4.4 Number of active browser tabs

This section analyzes the number of active browser tabs. A browser tab is considered active if some activities of a user is observed on the browser tab. We assume that the number of active browser tabs does not change before or after accessing ordinary URLs. However, it is observed that many malicious sites and some other sites open browser tabs unnecessarily. To confirm that, we analyzed the number of active browser tabs before and after malicious URL accesses.

Figure 5 shows a histogram of the number of active browser tabs within 10 minutes before and after malicious URL access. Over 99% of victims use 1–2 browser tabs before and after visiting malicious URLs. The figure also shows the number of active browser tabs within 10 minutes before a malicious access. Over 91% of victims use 1–4 browser tabs within 10 minutes before accessing malicious URLs⁴.

The difference in the number of active browser tabs between the above two histograms indicates that there could be browser tabs generated to reach malicious URLs, e.g., redirection pages opened in a new browser tab. This difference cannot be used as a feature for detecting malicious URL access because it requires access records after the malicious URL access, but it could be analyzed further to devise a

⁴Note that the maximum number of active browser tabs was 39; however, the frequency was negligible. Thus active browser tabs greater than 18 are not present in the histogram.

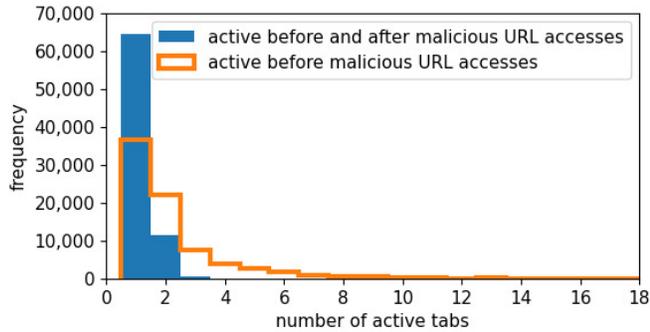


Figure 5: Histogram of the number of active browser tabs (February 2019–January 2020): browser tabs active within 10 minutes before and after malicious URL accesses and those before malicious URL accesses

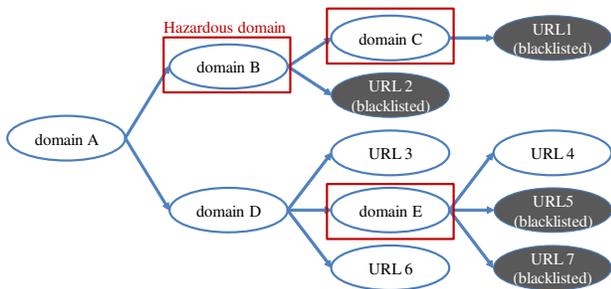


Figure 6: Concept of preemptive domain filtering scheme

feature to detect malicious URL access.

5 Preemptive Domain Filtering Scheme

The hazardous paths generated by the scheme proposed in Section 3 could be used to improve user protection. In this section, we propose a preemptive domain filtering scheme based on the risk level evaluation of domains on the hazardous paths.

Malicious URLs are included in blacklists; however, there are non-blacklisted URLs that lead to malicious URLs. We refer to domains that have high probability of leading to malicious URLs as hazardous domains. Access to malicious URLs can be minimized by identifying such domains and taking countermeasures, e.g., adding the domains to blacklists or alerting users. Figure 6 shows the concept of the proposed preemptive domain filtering scheme. In this figure, an access path tree consisting of multiple access paths that reach seven different URLs is described, and four of the URLs are included in a blacklist. The proposed scheme calculates the probability of reaching malicious URLs from a domain and identifies the domain as hazardous if the probability is greater than a certain threshold value. Accesses through domains B and C reach only blacklisted URLs; thus they are regarded

Table 7: Top 10 newly identified untrustworthy domains with a risk level above 50% (February 2019–January 2020)

Bookmarked domains	Access counts on hazardous paths	Access counts on all paths	Max risk level	Number of months
avgle.com	26,971	80,766	50.27	12
xbooks.to	1,185	11,138	66.66	4
codeday.me	991	1,105	100.00	11
ero-advertising.com	323	1,950	52.72	6
aphookkensidah.pro	272	1,043	100.00	12
tubepornclassic.com	109	782	77.14	5
highporn.net	105	152	69.07	1
erodoujin-index.net	82	109	75.22	1
fbk.tokyo	39	228	66.66	10
avli.me	38	58	65.51	1

Table 8: Top 10 newly identified untrustworthy domains with a risk level of 100% (February 2019–January 2020)

Bookmarked domains	Access counts on hazardous paths	Access counts on all paths	Number of months
codeday.me	991	1,105	11
aphookkensidah.pro	272	1,043	12
collectionanalyser.com	32	65	3
vidia.tv	27	33	3
livetotal.tv	27	104	4
jqaaa.com	27	28	3
eimusics.com	25	63	5
dentaint.pro	23	30	5
livetotal.net	22	52	4
javbraze.com	20	20	2

as hazardous domains. Accesses through domain E reach both blacklisted and non-blacklisted URLs; however this domain is regarded as hazardous because the probability is greater than the threshold value. Other domains are regarded as non-hazardous because the probability is less than the threshold value. These hazardous domains are not included in the blacklist; however, we could block access to these hazardous domains or alert users to minimize the risk of users' accessing malicious URLs.

To realize this preemptive domain filtering, we first reconstruct hazardous paths using the proposed scheme described in Section 3. We then extract all domains on the paths. Note that all domains on the path reach more than one malicious URL, while domains that did not appear on hazardous paths are not known to reach any malicious URLs. Finally, we analyze the risk levels of the URLs on hazardous paths using the risk level $R(\text{domain})$ defined in Section 4.2.

5.1 Identifying hazardous domains

We evaluated risk levels of all domains on hazardous paths using the proposed preemptive domain filtering scheme. Table 7 shows the top 10 newly identified domains with a risk level above 50%, and Table 8 shows those with a risk level of 100% over 12 months. Note, domains already identified by GSB have been excluded. The maximum risk level column shows the maximum value of monthly risk levels of a domain during the 12 months, while the number of months column shows the number of months the domain appeared on hazardous paths⁵. As can be seen, the proposed scheme can identify non-blacklisted domains that most likely navigate users to malicious URLs.

Table 9 shows the breakdown of the number of domains on hazardous paths by risk levels. To protect users, if the risk level of a domain is above a certain threshold, we could filter traffic on a domain or issue alerts. If the threshold is set to 80%, 355 domains are identified as hazardous domains in addition to 619 domains that have already been identified by GSB.

5.2 Blocked URLs

By enforcing the proposed preemptive domain filtering scheme and blocking access to hazardous domains, some URLs will become unreachable. To determine the effectiveness of the blocking, we identified URLs that would have been blocked if we have enforced the scheme using the dataset. Note that the scheme needs to know the threshold value to identify hazardous domains that need to be filtered as we have seen in Section 5.1, and we set the value to 80% in this section. URLs that became unreachable fall into one of the following types.

1. **Blacklisted URLs:** The proposed scheme blocks accesses to the hazardous domains that are located on the way to the blacklisted URLs. Therefore, users cannot reach the URLs, though access to these URLs are anyway blocked by the existing blacklists. Note that we used GSB for the blacklist, but other blacklists can be used in place of GSB to select domains for preemptive filtering.
2. **URLs included in other blacklists:** These were not blacklisted by GSB but were known to be malicious by the other blacklists. Here, we used a proprietary blacklist that identifies malicious URLs based on the signatures on URL strings. However, the number of these URLs remains small. We conjecture it is because policies of blacklists differ each other on what is detected as malicious. Indeed, GSB identifies more social-engineering type URLs and less malware-related URLs than the proprietary blacklist we used.

⁵Note that the GSB periodically revises the evaluation results of blacklisted entries.

3. **Non-blacklisted URLs whose domains are the same as the blacklisted URLs:** It may take some time for the GSB to register a malicious URL; therefore the URL may not be registered when a user visits it. These URLs will or should be blacklisted. Sometimes, its domain instead of the URL could be registered on the blacklist.
4. **Unreachable URLs:** These URLs are already unreachable at the time of writing this paper. Malicious URLs often disappear after a period of time. Therefore, the likelihood that these URLs are malicious is not too small.
5. **URLs with illegitimate or harmful content:** Many of the blocked URLs that do not fall into any of the above four categories deal with pornography, scanned manga and books, and music and video files. None of these URLs are listed in the Alexa Top 1,000 sites on a global basis [7] during the entire observation period. Although the proprietary blacklist we used did not recognize these URLs as malicious, determination of maliciousness largely depends on the policy of each blacklist, and these pages are not necessarily legitimate even if they are not included in the blacklist. These pages are likely to be irrelevant for the daily lives of most users. Therefore, the impact of those pages becoming unreachable is limited.

These URLs that were made unreachable are either malicious, unreachable, illegitimate, or harmful; thus blocking them would help improve user protection without impairing their legitimate activities.

6 Discussion and Analysis

Each technique has been discussed and evaluated in earlier sections. This section discusses issues related to our data collection and analysis approaches, limitation of our dataset, and directions for further analyses.

6.1 Advantages of user-side data collection

The uniqueness of our overall approach stems from our user-side data collection, which provides various data of browser users that cannot be collected on the network. These data provide two types of advantages: analytical efficiency and access to user-side data.

1. **Analytical efficiency:** The collected data includes user IDs and browser tab IDs. We can narrow down the data we need to look into by filtering with a user ID and a browser tab ID. In this way, the complexity and ambiguity of the data will be minimized, leading to more accurate analysis. As a result, the cost and time required for the analysis are also minimized.
2. **Access to user-side data:** We can obtain data that are unavailable from data collected on the network,

Table 9: Evaluation of domains on hazardous paths (February 2019–January 2020)

Risk level	Blacklist coverage	Number of domains												Total
		2019												
		Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	2020	
		Jan											Jan	
100%	full	51	58	60	72	61	289	28	44	50	74	69	47	903
	partial	1	2	3	1	1	3	0	1	1	2	0	1	15
	none	41	42	44	24	42	69	53	24	30	32	38	28	467
[80%-100%)	partial	0	2	1	1	2	2	1	0	0	3	1	2	15
	none	7	1	3	2	2	1	4	1	1	3	3	1	29
[60%-80%)	partial	3	2	1	2	0	6	4	0	1	0	2	1	22
	none	8	10	5	0	5	11	11	7	4	10	9	4	84
[40%-60%)	partial	0	1	1	1	0	6	0	0	0	1	1	1	12
	none	24	35	25	12	20	29	23	12	19	15	18	14	246
[20%-40%)	partial	3	0	2	3	1	3	2	0	1	1	0	1	17
	none	34	53	34	26	20	36	30	27	24	26	33	22	365
Others	partial	6	5	4	1	4	5	1	2	2	0	0	1	31
	none	333	398	218	192	193	250	250	160	192	183	163	173	2,705
Total		511	609	401	337	351	710	407	278	325	350	337	296	4,912

e.g., users' navigation information, which enables us to analyze users in detail. Indeed, our browser extension can collect data that are not listed in Section 2.1, such as the list of installed browser extensions and process information.

This paper demonstrated the usability of our data collection and analysis approach, and we hope this paper will encourage per-user data analysis to better protect users.

6.2 Attracting and Motivating Users

In this work, the primary is the number of users who install our browser extension and continue using it. The browser extension collects user's privacy-related information, which may reveal information they do not want anybody to know. Although the terms and conditions state that we do not link the data and the user's identity, this may discourage people from installing the browser extension.

To motivate users to install our browser extension, we implemented a campaign in the past, where a user can obtain a JPY 2,000 Amazon gift card. The campaign was successful from the standpoint of encouraging users to install the browser extension; however, over half of the users stopped using the browser extension within three months, indicating that the campaign was unable to motivate users to continue using the software.

Rather than asking people to install and use the browser extension, we redesigned the browser extension so that people would be interested in installing and continuing to use it by using a popular character, called Tachikoma, a popular character in the Ghost in the Shell universe [9]. By continuing to activate the browser extension, users can see Tachikoma in their browser. People who like the story or the character are

motivated to install the browser extension and continue using it. We then prepared a web page where people can download and install the browser extension. We also advertised our data collection activities at large IT events. As can be seen in Figure 1, the current approach was successful from the standpoint of maintaining the number of active users, i.e., motivating users to continue using the browser extension.

6.3 Limitation of our dataset

The dataset we used has some limitations. When conducting further analysis, these limitations should be considered.

First, the demographics of the users of the browser extension are limited because we are distributing the browser extension to people who like the Tachikoma character. People who like Tachikoma tend to be familiar with IT; thus their use of the web does not necessarily accurately represent behavior of general web users. In addition, the browser extension and its distribution page are only available in Japanese; thus the data analysis will not reflect the behavior of global web users.

Second, depending on the purpose of analysis, the method used to label malicious URLs needs to be reviewed. We used GSB entries to flag malicious URLs; however, this is not always desirable. Depending on the policies, users may implement different blacklists.

Third, the scale of the dataset is limited. Although the number of browser extension users was sufficient for the analysis in this paper, it is very small considering the number of web users in general. In particular, the number of victims is too small. When conducting other types of analysis, such as user classification, the scale of the data can be insufficient, and one such example is discussed in Section 6.4. To cope with this issue, measures such as effective campaigns to attract

more users should be devised and deployed. Future works could consider these limitations of the dataset.

6.4 Further Analyses

This paper demonstrated the usefulness of analyzing user-side data collected through browsers. Further analyses are encouraged to deepen the understanding of user behaviors, including the analysis of access records before session reconstruction as discussed in Section 3.2. In this section, two more analysis directions are shown below.

We may use data that was unused in this paper to reconstruct hazardous paths in detail. For example, transition qualifiers, which can be collected through the `chrome.webNavigation` API, could be used to deepen the understanding of user behaviors. The API provides four transition qualifiers: "client_redirect," "server_redirect," "forward_back," and "from_address_bar." Their usability is demonstrated in a case, where a user browses pages in the following manner. (1) A user visits a search engine result page (whose transition type is set to "form_submit"). (2) The user clicks on one of the links on the page. (3) The user pushes the "back" button on the browser. In this case, the access record of the access (1) is used as the access record for the access (3), meaning that its transition type is "start_page" and the referer does not comply with access (2). In this study, the access path reconstruction scheme judges that the access (3) is the path entry point; however, we could trace back further by considering transition qualifier information.

Moreover, user behaviors can be analysed at a finer granularity. For example, user behaviors can be analyzed for each type of detected threats rather than using a binary label, i.e., malicious or not malicious. Indeed GSB provides types of detected threats, e.g., "MALWARE" and "SOCIAL_ENGINEERING"; however, we could not use these in this paper because our dataset did not have a sufficient amount of access records for each type of the threats detected by GSB. As discussed in Section 6.3, our dataset was too small to analyze accesses based on these types.

Various other analyses can be conducted for different purposes. These analyses will aid in building efficient schemes to improve user protection.

7 Related Work

Various studies have been reported in the area of malicious URL analysis. They take different approaches with different datasets. This section introduces major such works.

Previous studies have analyzed web page content to identify malicious sites [10–17]. A JavaScript code analysis at a bytecode level has been proposed [11] to cope with the obfuscation. Another study proposed a link structure analysis technique [12] to detect compromised websites by identifying structural anomalies. In addition, a cascading style sheets

analysis technique has been proposed [13] to detect pages leading to malware downloads.

Lexical analysis has also been proposed to extract features from URL strings and identify malicious sites [18–23]. Among studies that apply lexical analysis, one study [19] attempts to achieve online learning; thus it does not use information that requires time to obtain. That study uses the URL string and host-related information, i.e., host name, primary domain, TLD, whois information, AS number, and geographical information, as features.

In addition, several studies have focused on building and analyzing redirection chains [24–29], which are often observed when users reach malicious URLs. For example, the SpiderWeb system [28] analyzes HTTP redirection chains. It uses five types of features, i.e., client, referer, landing page, final page, and redirection graph, to distinguish chains that correspond to malicious activity and those that are legitimate. WarningBird [29] detects malicious URLs posted on Twitter by analyzing the correlations of redirection chains, while Surf [25] identifies redirects to malicious URLs that are originated from search engine results.

Access paths followed by users who eventually fall victim to different types of malware download attacks, called malware download paths, have also been analyzed [30]. In that study, the authors proposed a download path traceback technique as well as a technique to determine whether the path is social engineering or drive-by, using various features, such as domain ages and the number of hops to exploit pages. Another study focuses on social engineering URLs [31]. That study uses a random forest algorithm to determine the occurrence of a social engineering attack from ad-related sites by learning about such past attacks using features extracted from the download paths. In addition, research focusing on identifying malicious exploit kits has also been reported [32].

Other studies have focused on user behavior [33–37]. A study analyzes traffic on a mobile cellular network to predict whether a user will visit a malicious URL within a month based on past browsing activities and a questionnaire [34]. The study also predicts whether a user would access a malicious URL within a session based on information from past records in the same session.

Various other studies have been reported in this area, including domain reputation systems [38–40] and signature generation techniques [41, 42]. Contrary to these, we collected data at the user side, which enabled us to analyze the user activities in detail by discerning users and browser tabs. We also proposed a preemptive domain filtering scheme that identified hazardous domains that were not included in blacklists.

8 Conclusion

Our user-side web access record collection approach enabled us to access to a wide range of data, such as user IDs, browser

tab IDs, and user navigation information, which facilitated efficient and detailed analysis of user behavior. We have reconstructed hazardous paths from the collected data by continuously tracing previous access records until we identify entry points of the paths. The hazardous path reconstruction was efficient because it was able to discern users and browser tabs. Then, we analyzed the reconstructed hazardous paths to deepen the understanding of users' web browsing activities and revealed several analysis results, including that bookmarks are the major entry points of hazardous paths. It indicated that sanitizing bookmark entries will minimize the risk of accessing malicious URLs. Furthermore, we proposed a preemptive domain filtering scheme that identifies and filters domains that lead to malicious URLs. The effectiveness of the proposed scheme was demonstrated by revealing non-blacklisted domains that ultimately led users to malicious URLs. We hope that our work in this paper will contribute to the security of the web.

Acknowledgment

This study has been supported by WarpDrive project [2]. We thank all the colleagues of the project. We would like to extend our gratitude to our paper shepherd, Amin Kharraz, and the anonymous reviewers for their feedback and assistance.

References

- [1] Google Safe Browsing. <https://safebrowsing.google.com/>. Accessed: June 1, 2020.
- [2] WarpDrive. <https://warpdrive-project.jp/>. Accessed: June 1, 2020.
- [3] Chrome: developer. <https://developer.chrome.com/home>. Accessed: June 1, 2020.
- [4] chrome.webRequest. <https://developer.chrome.com/extensions/webRequest>. Accessed: June 1, 2020.
- [5] chrome.webNavigation. <https://developer.chrome.com/extensions/webNavigation>. Accessed: June 1, 2020.
- [6] chrome.history. <https://developer.chrome.com/extensions/history>. Accessed: June 1, 2020.
- [7] Alexa top sites. <https://www.alexa.com/topsites>. Accessed: June 1, 2020.
- [8] VirusTotal. <https://www.virustotal.com/>. Accessed: June 1, 2020.
- [9] Ghost in the Shell. http://www.production-ig.com/contents/works_sp/16_/index.html. Accessed: June 1, 2020.
- [10] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, 2011.
- [11] Y. Fang, C. Huang, L. Liu, and M. Xue. Research on malicious javascript detection technology based on lstm. *IEEE Access*, 6, 2018.
- [12] P. Ravi Kumar, P. Herbert Raj, and P. Jelciana. A framework to detect compromised websites using link structure anomalies. In *Computational Intelligence in Information Systems*, 2019.
- [13] B. Chen, and Y. Shi. Malicious hidden redirect attack web page detection based on css features. In *2018 IEEE 4th International Conference on Computer and Communications*, 2018.
- [14] B. Altay, T. Dokeroglu, and A. Cosar. Context-sensitive and keyword density-based supervised machine learning techniques for malicious webpage detection. *Soft Comput.*, 23, 2019.
- [15] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.
- [16] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.
- [17] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.
- [18] M. Darling, G. Heileman, G. Gressel, A. Ashok, and P. Poornachandran. A lexical approach for classifying malicious urls. In *2015 International Conference on High Performance Computing Simulation*, 2015.
- [19] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Learning to detect malicious urls. *ACM Trans. Intell. Syst. Technol.*, 2, 2011.
- [20] R. Verma, and K. Dyer. On the character of phishing urls: Accurate and robust statistical learning classifiers. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.
- [21] A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In *2011 Proceedings IEEE INFOCOM*, 2011.

- [22] D. Huang, K. Xu, and J. Pei. Malicious url detection by dynamically mining patterns without pre-defined elements. *World Wide Web*, 17, 2014.
- [23] G. Tan, P. Zhang, Q. Liu, X. Liu, C. Zhu, and L. Guo. Malfilter: A lightweight real-time malicious url filtering system in large-scale networks. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications*, 2018.
- [24] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *2013 IEEE Symposium on Security and Privacy*, 2013.
- [25] L. Lu, R. Perdisci, and W. Lee. Surf: Detecting and measuring search poisoning. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.
- [26] H. Mekky, R. Torres, Z. Zhang, S. Saha, and A. Nucci. Detecting malicious http redirections using trees of user browsing activity. In *IEEE Conference on Computer Communications*, 2014.
- [27] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.
- [28] G. Stringhini, C. Kruegel, and G. Vigna. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013.
- [29] S. Lee, and J. Kim. Warningbird: A near real-time detection system for suspicious urls in twitter stream. *IEEE Transactions on Dependable and Secure Computing*, 10, 2013.
- [30] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Webwitness: Investigating, categorizing, and mitigating malware download paths. In *24th USENIX Security Symposium*, 2015.
- [31] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad. Towards measuring and mitigating social engineering software download attacks. In *25th USENIX Security Symposium*, 2016.
- [32] T. Taylor, X. Hu, T. Wang, J. Jang, M. P. Stoecklin, F. Monrose, and R. Sailer. Detecting malicious exploit kits using tree-based similarity searches. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016.
- [33] D. Canali, L. Bilge, and D. Balzarotti. On the effectiveness of risk prediction based on users browsing behavior. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, 2014.
- [34] M. Sharif, J. Urakawa, N. Christin, A. Kubota, and A. Yamada. Predicting impending exposure to malicious content from user behavior. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.
- [35] F. L. Lévesque, J. M. Fernandez, and A. Somayaji. Risk prediction of malware victimization based on user behavior. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, 2014.
- [36] M. Ovelgönne, T. Dumitraş, B. A. Prakash, V. S. Subrahmanian, and B. Wang. Understanding the relationship between human behavior and susceptibility to cyber attacks: A data-driven approach. *ACM Trans. Intell. Syst. Technol.*, 8, 2017.
- [37] Y. Carlinet, L. Mé, H. Debar, and Y. Gourhant. Analysis of computer infection risk factors based on customer network usage. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*, 2008.
- [38] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster. Building a dynamic reputation system for dns. In *Proceedings of the 19th USENIX Conference on Security*, 2010.
- [39] M. Antonakakis, R. Perdisci, W. Lee, N. Vasiloglou, and D. Dagon. Detecting malware domains at the upper dns hierarchy. In *Proceedings of the 20th USENIX Conference on Security*, 2011.
- [40] M. A. Rajab, L. Ballard, N. Lutz, P. Mavrommatis, and N. Provos. CAMP: content-agnostic malware protection. In *20th Annual Network and Distributed System Security Symposium*, 2013.
- [41] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee. Arrow: Generating signatures to detect drive-by downloads. In *Proceedings of the 20th International Conference on World Wide Web*, 2011.
- [42] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

What’s in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques

Ahmet Salih Buyukkayhan
Microsoft

Can Gemicioglu
Northeastern University

Tobias Lauinger
New York University

Alina Oprea
Northeastern University

William Robertson
Northeastern University

Engin Kirda
Northeastern University

Abstract

Cross-Site Scripting (XSS) is one of the most prevalent vulnerabilities on the Web. While exploitation techniques are publicly documented, to date there is no study of how frequently each technique is used in the wild. In this paper, we conduct a longitudinal study of 134 k reflected server XSS exploits submitted to XSSED and OPENBUGBOUNTY, two vulnerability databases collectively spanning a time period of nearly ten years. We use a combination of static and dynamic analysis techniques to identify the portion of each archived server response that contains the exploit, execute it in a sandboxed analysis environment, and detect the exploitation techniques used. We categorize the exploits based on the exploitation techniques used and generate common exploit patterns. We find that most exploits are relatively simple, but there is a moderate trend of increased sophistication over time. For example, as automated XSS defenses evolve, direct code execution with `<script>` is declining in favour of indirect execution triggered by event handlers in conjunction with other tags, such as `<svg onload`. We release our annotated data, enabling researchers to create diverse exploit samples for model training or system evaluation.

1 Introduction

Vulnerability reward programmes have emerged as a way of encouraging and coordinating the discovery and disclosure of vulnerabilities by independent researchers [7, 18, 21]. As a popular example, the commercial platform HackerOne [9] partners with software vendors and administers bug bounty programmes on their behalf. Non-commercial platforms such as XSSED [6] and OPENBUGBOUNTY [16], which specialise in Web vulnerabilities, are *one-sided* [18] in the sense that they allow submissions of vulnerabilities affecting any website instead of restricting them to partner organisations.

Bug bounty programmes and platforms have been the subject of a large number of studies, which to date have nearly exclusively focused on the economics of vulnerability discovery such as incentives and return on investment [7, 21], as

well as the user population, affected websites, patching delays, and high-level categorisation of vulnerability types [18, 21]. However, little is known in terms of technical details about the exploits submitted by users.

In this paper, we perform a longitudinal study of the exploitation techniques used by the authors of reflected server XSS exploits in XSSED and OPENBUGBOUNTY over a period of nearly ten years. This required us to solve a number of technical challenges. The exploits submitted to the two platforms are not available in isolation, but must be extracted from the archived request and server response. This is particularly difficult, given that we have no insight into the inner workings of the web application, and no access to a comparable, “clean” server response without the exploit. To this end, we design techniques to isolate the reflected exploit by comparing the request and response data, and accounting for server transformations such as encoding or escaping.

Once the attack string is isolated, we need to extract features for exploitation techniques. Some techniques, such as syntax tricks attempting to confuse filters in web apps and firewalls by exploiting a parsing disparity, are not visible when a page is opened in a browser. We detect this class of techniques statically and comprehensively validate our method.

Other features are visible only at runtime, such as call stacks, or whether the injected code actually works. In fact, many archived server responses contain multiple instances of reflected code, but input sanitisation appears to be applied inconsistently, causing some instances to execute while others do not. We collect this information by dynamically executing archived exploits. Often, exploits have environmental dependencies or require certain events to be triggered before they can execute, thus we developed a sandboxed environment with lightweight browser instrumentation that can simulate different network conditions and user interaction.

For our analysis, we integrate the data collected by the static and dynamic analysis methods. When the results of both static and dynamic analysis are available, we detect the exploitation techniques with 100% true positive rate. Our methodology enables us to perform a comprehensive longitud-

inal study of 134 k reflected server XSS exploits. Surprisingly, a large percentage of exploits, 65.0 % in XSSED and 11.7 % in OPENBUGBOUNTY, use no technique beyond inserting a `<script>` tag after possibly closing the previous tag, and make no attempt to bypass filters.

We also analyse common exploit patterns, defined as a collection of techniques used in combination. We find that the most common pattern accounting for about half of all exploits in XSSED is closing the previous tag and inserting a `<script>` tag with the payload. Presumably due to increased filtering of this tag, in the more recent submissions to OPENBUGBOUNTY, we observe a trend of replacing the `<script>` tag with `` or `<svg>` tags and indirect code execution using the `onload` and `onerror` event handlers.

Our analysis sheds light on XSS exploitation trends and popular exploit patterns. We also discuss how exploitation techniques evolve over time, as automated defenses are implemented by modern browsers. To summarise, this paper makes the following contributions:

- We demonstrate that it is possible to accurately identify proof-of-concept exploits in archived web pages. We implement a system to execute exploits and extract features with a unified static and dynamic approach.
- We analyse ten years of reflected server XSS exploits to quantify the use of various exploitation techniques.
- We show that there is only a moderate change in techniques and sophistication despite the long time span, supporting the hypothesis of an “endless” supply of low-complexity vulnerabilities on the Web, and suggesting a lack of incentives for users of the platforms to find and contribute more sophisticated XSS vulnerabilities.
- We release the dataset [1] used in this paper.

2 Background

Cross-Site Scripting (XSS) is a class of attacks that consist in injecting attacker-controlled JavaScript code into vulnerable websites. The attack targets the visitors of a compromised website. To the victim’s browser, the injected code appears to originate from the compromised website, thus the Same Origin Policy does not apply and the browser interprets the code in the context of the website. Consequently, attackers can exfiltrate sensitive information, or impersonate the victim and initiate transactions such as purchases or money transfers.

This study focuses on reflected server XSS in which the vulnerability arises from server-side templating and the inability of the browser to distinguish the trusted template from the untrusted user input. To prevent attacks, the developer must escape sensitive characters in the user input. However, which characters are sensitive and how they can be escaped depends on the sink context. For example, inside a JavaScript string

context, quotes must be escaped as `\`, inside HTML tags as the HTML entity `"`; and outside of HTML tags, they do not have a special meaning. General-purpose server-side programming languages typically do not escape user input since they are unaware of the sink context. Furthermore, in some cases developers may wish to allow certain types of markup in user input, such as style-related tags in articles or blog posts submitted by users. Unfortunately, developers often omit input sanitisation entirely, use an incorrect type of escaping, or implement custom sanitisation code that is not sufficient to block the attacks. For example, developers who would like to allow certain tags but not script may remove the string `"script"` from user input, but they may fail to account for case differences (`<ScRiPt>`) or indirect execution through event handlers (`onload="alert(1)"`). Web Application Firewalls are often implemented using regular expressions and may be vulnerable to similar issues [4, 13].

Cross-Site Scripting exploits bugs in websites as opposed to browsers. As such, XSS is not to be confused with JavaScript-based attacks that exploit browser bugs to cause memory corruption and gain code execution at the operating system level. However, XSS could be used as a vehicle to inject such types of payloads into websites. In this work, we do not study such malicious payloads; rather, we focus on the tricks that authors of XSS exploits use to bypass filters up to the point where they gain script execution capabilities. The two datasets that we use, XSSED and OPENBUGBOUNTY, contain only benign proof-of-concept (PoC) payloads such as `alert(1)`, but exploits do differ in how they achieve code execution.

2.1 Vulnerability Rewards and Platforms

Bug bounty platforms provide a formalised way for bug hunters and website operators to interact. Furthermore, they often provide cash rewards for verified bug reports. Economic aspects of vulnerability rewards have been studied extensively, such as for Chrome and Firefox by Finifter et al. [7], or for the HACKERONE and WOYUN platforms by Zhao et al. [21]. The arguably most well-known platform, HACKERONE, operates vulnerability reward programmes on behalf of other organisations. The platform is “closed” in the sense that vulnerabilities can be submitted only for websites or software developed by participating organisations. Furthermore, those organisations are typically responsible for verifying vulnerability reports, determining the amount of a potential reward, and deciding on whether to publish the vulnerability.

In contrast to closed commercial platforms, our research uses data obtained from two open, non-profit bug bounty websites specialised in XSS. In this model, users may submit exploits targeting any website. The first of the two platforms, XSSED [6], was founded in 2007 and appears to be largely dormant since 2012. The second platform, OPENBUGBOUNTY [16] was launched in 2014 under the name XSSposed, and is still active at the time of writing.

2.2 Related Work

Prior studies on vulnerability reward programmes have focused on economic considerations (e.g., [7, 18, 21]). Zhao et al. [21] dedicated a small part of their study to the submitted exploits, but these encompassed many different vulnerability classes, and the analysis was restricted to a few high-level metrics such as “severity” provided by the WOYUN platform. Furthermore, the authors noted that “Wooyun may ignore vulnerabilities that are considered irrelevant or of very low importance, such as many reflected XSS vulnerabilities.” In their analysis of OPENBUGBOUNTY, Ruohonen and Alodi [18] considered platform metrics such as patching delays and user productivity. Instead of these general-purpose exploit or platform metrics, we conduct a deeper analysis of the different techniques used specifically in reflected server XSS.

Cross-Site Scripting has been considered in a large body of scholarly research, mostly from a vulnerability detection and exploitation prevention perspective. However, we are aware of only very few works that quantify the occurrence of different XSS exploitation techniques. For example, while evaluating their proposed XSS filter, Bates et al. [3] classified the sink contexts of 145 random exploits from XSSSED. Our findings in Section 4.2 are in line with theirs, and provide additional detail about unnecessary context escaping, and multiple or non-executing reflections in server responses. Furthermore, we work with archived server responses, whereas Bates et al. could only consider live pages that were still vulnerable.

In DOM-based client XSS, several works [14, 15, 20] reported characteristics of exploitable data flows, such as the source and sink types, and cognitive complexity. In our context of reflected server XSS, data flows and transformations occur in (hidden) server-side code, thus we need different methods. Furthermore, the goal of the analysis was to characterise the “root causes” of vulnerabilities. In contrast, we use human-curated data sets to quantify different exploit techniques.

Closest to our work is a study by Scholte et al. [19] of 2,632 XSS and SQL injection attacks found in the National Vulnerability Database. The authors measured the complexity of XSS exploits using five static features such as having encoded characters or event handlers. Scholte et al. found that vulnerabilities were simple, and their complexity did not increase. Our results confirm the trends reported in 2011. We extend them in depth, scale, and time span by using a fifty times larger data set, extracting five times more static features, adding dynamic analysis, and characterising exploit authors.

Similar in spirit, but different in the covered environment, are two studies analysing Java exploits [12] and malware [5].

3 Methodology

In this work, we quantify the techniques used by the authors of XSS exploits archived in XSSSED and OPENBUGBOUNTY. This required us to solve many challenges such as locating

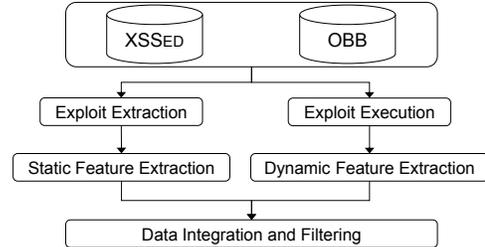


Figure 1: Overview of our static and dynamic analysis system.

Database	Date Range	Authors	Submissions	Websites
XSSSED	2007-01-24 – 2015-03-12	2,579	43,939	33,747
OBB	2014-06-18 – 2017-04-11	980	119,946	86,339

Table 1: Raw XSS data collected (before filtering).

attack strings embedded in HTML responses, triggering and executing the attack in a controlled environment, and identifying XSS techniques used. We address them by combining static and dynamic analysis methods, as outlined in Figure 1.

3.1 Exploitation Techniques

Reflected server XSS exploits can use a range of techniques to bypass incomplete input sanitisation and achieve code execution. Various cheatsheets [2, 8, 10, 11] collect these techniques. Our work models portable techniques in the sense that they work in most modern browsers rather than in specific browsers or versions. We also exclude techniques that we implemented based on the cheatsheets, but that appear rarely in our datasets. On the other hand, we include additional techniques that we observed while working with our datasets. Table 4 on page 10 lists all exploitation techniques considered in this paper, grouped into several categories.

3.2 Data Collection

Our analysis of XSS exploits aims to cover a broad range of techniques and study how they evolve over time. To do so, we extracted all XSS exploits publicly disclosed on XSSSED and OPENBUGBOUNTY until April 2017, as shown in Table 1. This represents the latest available dataset, as XSSSED is now largely dormant, and OPENBUGBOUNTY has implemented anti-crawl measures. The operators of the databases claim that all entries were confirmed to work at the time of submission.

3.3 Exploit String Extraction

XSSSED and OPENBUGBOUNTY archive XSS vulnerabilities by storing the request data (URL, cookies, POST data) and the HTML page returned by the server. Therefore, first we must isolate the attack string embedded in request and response. Prior work found that nearly half of sampled XSSSED entries

had been fixed or the affected website was not reachable [3]. Instead, we extract XSS exploits by looking for reflected strings that exist both in the request and the response.

Since nearly all exploits only show a message using the methods `alert()`, `prompt()` or `confirm()`, these method names typically appear in the URL. In case of obfuscation, the URL contains other strings, such as `eval()` or related method names. Thus, for the limited purpose of locating these payloads, we search the request and response for these keywords.

To extract attack strings, we implemented a greedy heuristic that expands the matching range from the keyword to the left and right as long as the character sequences match. Servers can transform the inputs before reflecting them, such as encoding characters, truncating or expanding the string. Without knowledge of the server-side logic, we cannot solve this issue in a generally sound way. However, we do support the most common transformations, specifically HTML entity, Unicode, URL, and certain cases of double URL encoding, and our matching is case-insensitive. Additionally, we allow up to two consecutive non-matching characters, provided that there are more matching characters afterwards. Our validation shows that this works well in the context of our dataset.

One server response may contain multiple matches when one injected parameter is reflected multiple times, or when exploits are injected into multiple parts of the URL. We initially match all reflection combinations, but eventually select a single pair that has been confirmed to be functional, and is unlikely to be a false positive (Sections 3.5 and 3.7).

3.4 Static Feature Extraction

Once the attack string has been isolated, we need to detect how it achieves JavaScript code execution. Especially syntax-based tricks such as whitespace or comment insertion, or non-standard syntax are not visible from the JavaScript runtime, as the browser parser converts the non-standard textual HTML into a canonical DOM representation. Given that our feature extraction operates on a previously known dataset, we implement it using regular expressions and iteratively refine and validate our implementation through manual labelling.

The techniques we detect statically (marked ○ and ● in Table 4, on page 10) include escaping from the injection sink context by ending string literals or closing a tag, case tricks such as `<ScRiPt>`, and using a slash instead of a space, as in `<svg/onload`. Additionally, we extract features such as event handlers so that we can combine them with dynamically detected features to reduce false positives. Depending on the technique, we make multiple attempts to match the respective regular expression, such as before and after decoding encoded sequences, or removing whitespace.

	Submissions		Messages TP / FP	
	XSSSED	OBB	XSSSED	OBB
No Trigger	38,009	94,693	38,926 / 188	95,779 / 286
Mouse Move	289	3,959	288 / 5	3,994 / 15
Mouse Click	145	103	123 / 23	97 / 7
Network Error	0	4	0 / 0	4 / 0

Table 2: Exploits executed by triggering events.

3.5 Exploit Execution

Almost two thirds of archived server responses in XSSSED and OPENBUGBOUNTY contain multiple reflections of attack strings. Some of them appear to be sanitised or truncated, preventing execution. In order to extract XSS techniques only from attack strings that execute, we need to run the exploits.

To execute exploits, we load the archived response pages in Chrome and Firefox. We add instrumentation to detect whether the exploit is working, and sandbox network traffic by serving all requests from an isolated local proxy. As for the static exploit string extraction, we leverage the fact that nearly all exploits contain a simple payload showing a message with `alert()`, `prompt()` or `confirm()`, and use the calling of any such method as a sign for a successfully executed exploit.

When simply opened in our sandbox environment, many pages do not result in the exploit being executed. In some cases, exploits require user interaction. In other cases, exploits only execute when external resources such as images or scripts are loaded successfully, or when resources fail to load. Unfortunately, neither XSSSED nor OPENBUGBOUNTY archive these resources, thus we replace them with generic versions. Our system attempts to execute each exploit in up to four rounds, as summarised in Table 2. In the first round, the proxy returns the archived page and then answers each resource request with an empty response and the HTTP 200 OK status code. For exploits not successfully executed, in the second round we move the mouse over all elements in order to trigger user interaction event handlers, and wait for ten seconds to allow delayed execution using `setTimeout()`. This results in 4.2% additional executions in OPENBUGBOUNTY. In the third round, we click on all elements, and in the fourth round, the proxy answers resource requests with empty documents and the HTTP 404 Not Found status code to trigger the `onerror` event.

During our attempts, we may accidentally trigger unrelated code in the page. When clicking a button, for instance, the page may display an error message. To distinguish page-related dialogues from alerts caused by injected exploit code, we semi-automatically classify the displayed messages. Messages displayed by exploits are often very similar, such as “XSSSED” or “OBB,” or the name of the author. We manually looked through the top 30 message texts in each database, which account for 70.3% of messages in XSSSED, and 99.7%

in OPENBUGBOUNTY. We confirmed that 58 of the 60 messages were related to exploits. For the remaining messages, we built a dictionary of attack-related keywords, such as the names of exploit authors commonly appearing in our data. When a message text contains none of these exploit keywords, and is not part of the 58 most frequent messages labelled as exploit-related, we assume it is a false positive and discard it.

For validation, we picked two random samples of 100 unique messages that were considered exploit-related and not in the top 30. In XSSSED, all sampled messages were confirmed based on the source code to originate from an injected exploit. In OPENBUGBOUNTY, our sample had a 2% false positive rate. Yet, as OPENBUGBOUNTY contains only 324 distinct messages with attack keywords beyond the top 30, we estimate a total of 7 distinct false positive messages, which corresponds to 0.007% of all message instances.

During our experiments, we found minimal differences between Chrome and Firefox. Around 0.3% of submissions in XSSSED, and 1.0% in OPENBUGBOUNTY worked only in Firefox. On the other hand, 2.4% of XSSSED, and 3.9% of OPENBUGBOUNTY worked only in Chrome. This disparity is mainly due to technical differences in our instrumentation that may prevent data extraction in corner cases such as pages crashing the browser. Browser families also differ in how they handle (partially) broken pages, and in which exploitation techniques they “support.” Chrome executes more exploits than Firefox, and returns richer metadata that we require to combine static and dynamic features. When using Firefox, the lack of metadata allows combining features for only 23% of OPENBUGBOUNTY submissions, as opposed to 83% with Chrome. For simplicity, we only report Chrome results.

3.6 Dynamic Feature Extraction

While executing exploits, our instrumentation detects the use of certain exploitation techniques. For some, detection is only practical at runtime, such as exploit code interacting with page code, or assignment of the `alert` method to a variable and subsequent calling of the variable. We also extract a number of features in parallel to the static feature detection to reduce false positives. This includes the use of obfuscation or quote avoidance methods such as `eval()` or regular expressions.

From an implementation point of view, our browser extension injects a content script that intercepts calls to a range of methods and property accesses. It logs the values and types of parameters such as evaluated strings or message texts, and records the stack trace of the call. We extract line numbers and offsets to distinguish features detected in exploits from detections in unrelated page functionality (Section 3.7).

Due to our choice of a light-weight browser instrumentation, we do not detect certain corner cases. These include exploit code in `data:` or `javascript:` URLs or an `<iframe>`, use of `innerHTML` or `outerHTML`, and element creation using the DOM API `document.createElement()`. Some of

these limitations are due to incorrect line number and offset values returned by the browser, which prevents us from combining these dynamically detected features with their static detection counterpart. Among all features, only page code interaction (F6) is exclusively based on dynamic data. In a random sample of 25 out of 145 exploits where this feature was detected, we did not observe any false positives.

3.7 Data Integration and Filtering

We combine the data collected during the static and dynamic analysis to address the shortcomings of each individual approach. By using only exploits that were detected both statically and dynamically, we exclude exploit reflections that do not execute, or other false positives of static detection. By restricting dynamic detections to source code ranges statically detected as the attack string, we exclude false positives of dynamic detection due to code unrelated to the exploit.

The static extraction returns byte ranges of the reflected data. From the dynamic execution, we obtain stack traces of method invocations with line number and offset values. At a high level, integrating static and dynamic data means checking whether the dynamic line number and offset are within the static exploit range. Our implementation adds three preparatory steps. First, we extract the character encoding used by the browser so that we can correctly compare offsets involving multi-byte characters. Second, we normalise endline characters in the server responses. Third, in several special cases, the offsets returned by Chrome do not point to the actual beginning of the respective statement but are shifted by the length of a syntactical construct or point to the end of the tag, and need to be adjusted accordingly.

Almost two thirds of server responses reflect the injected exploit multiple times. When matching static and dynamic data, we combine all pairs and select a single exploit to represent each HTML page. We choose as the representative exploit one where the message is a known exploit text, and which is the shortest string among the matches with the most frequent set of exploit features detected on the page. Only 1.6% of submissions had more than one working reflection and more than one unique feature set, thus in the vast majority of cases, our selection does not make any difference.

Similarly to the exploits, we also combine features by restricting them to cases where both the static and the dynamic occurrence was observed. These are marked as ● in Table 4. We validated a subset of important features, shown in Table 3, by labelling 25 random exploits from three detection categories: both static and dynamic data were combined ($S \cap D$), only static features ($S - D$), or only dynamic features ($D - S$) were detected. Features with only static detection are either false positives of static detection, or false negatives of dynamic detection, and vice versa for features with only dynamic detection. The results show that our conservative approach resulted in no false positives in feature detections. Detections made by

<i>S</i> : static, <i>D</i> : dynamic (% of all detections)	<i>S</i> − <i>D</i>			<i>D</i> − <i>S</i>			<i>S</i> ∩ <i>D</i>		
	%	TN	FN	%	TN	FN	%	TP	FP
I1: document.write	8.5	10	1	0.8	1	0	90.7	25	0
I2, I3: eval, setImmediate	11.5	24	1	1.8	9	0	86.7	25	0
O1: char. code to str.	1.2	25	0	0.0	0	1	98.8	25	0
O2: regular expr.	0.1	21	4	0.3	24	1	99.6	25	0

True/false negatives/positives in samples of 25 detections (if available).

Table 3: Validation of Static and Dynamic feature extraction.

only one type of analysis sometimes contain false negatives, but the combination of static and dynamic analysis results in perfect accuracy.

3.8 Validation

We successfully combine static and dynamic data for 36,229 XSS submissions in XSSED (82 %), and 97,677 XSS submissions in OPENBUGBOUNTY (83 %). In the remaining 17.1 % of submissions, we cannot match a statically extracted attack string in the server response with a dynamically observed attack message, thus we exclude these exploit submissions from further analysis. To further investigate, we manually labelled a random sample of 1,000 such cases. The most frequent reason for exclusion was the inability to execute the exploit (37.7 % of cases, or 6.4 % of the entire dataset). These cases were apparently caused by the use of nonstandard syntax not supported by Chrome, missing dependencies or event triggers, or the exploit not executing before our 10 second timeout. Exploits that fingerprint the environment and stop execution to prevent analysis would also fall into this category, but we did not observe any example in our data. Another 22.1 % of excluded submissions in the two databases contained incomplete data, or an unsuitable format such as screenshot submissions that we could not process. Around 0.3 % of excluded submissions were false negatives due to line number and offset matching issues. The remaining 39.9 % of excluded submissions (6.8 % of the entire dataset) were attack types that our analysis framework does not support. We do not further analyse exploits injecting references to remote JavaScript files (2.9 % of the entire dataset), as it is difficult to distinguish benign from malicious script URLs, and Chrome returns line numbers that are ambiguous when multiple such remote scripts are injected into the same page. Similar line number issues exist with exploits contained in `data:` or `javascript:` URIs (found in 0.4 % of all submissions). Around 0.2 % of the exploits redirect to a different page, and 0.9 % inject text into the HTML page instead of showing a message, causing their exclusion. Furthermore, we exclude exploits that are not reflected server XSS, as they imply the absence of the exploit from the URL (stored server XSS: 0.05 % of all submissions) or from the server response (DOM-based client XSS: 1.9 %), thus preventing us from isolating the attack string. Lastly, we exclude 0.09 % of the dataset because these submissions ap-

pear to involve techniques unrelated to HTML and JavaScript, such as SQL injection.

3.9 Limitations

Our datasets contain potential biases due to their open nature. The data may be biased towards simpler exploits, favour the use of identical exploits on multiple websites, and result in fewer submissions per website while covering several orders of magnitude more websites than closed platforms. Furthermore, we may under-report exploits for sites that operate their own (cash-based) bug bounty programme, as users may be motivated to submit their findings directly to that programme instead of reporting them to XSSED or OPENBUGBOUNTY. Because only 10.1 % and 13.3 % of websites in XSSED and OPENBUGBOUNTY, respectively, have multiple submitted exploits, our dataset should not be used to derive findings about the security posture of individual websites. There is no guarantee that users find vulnerabilities on a given website, that they submit them to one of the two databases covered in our study, and that the submitted exploit is as simple as the vulnerability allows. These limitations notwithstanding, we believe that our dataset (134k exploits submitted by 3k authors) gives us a unique insight into the XSS techniques used by a wide range of security testers.

Our method has some limitations. The detection being based on keywords, we might fail to locate exploits when none of these keywords is used in the exploit. Since we expect that every database entry contains an exploit, we can estimate the false negative rate through manual validation of entries without a match (Section 3.8). Our approach does not support “split” exploits with multiple cooperating parts that are injected separately, and would only detect the part with the keyword. Furthermore, static features fail to indicate techniques hidden under an obfuscation layer. However, our datasets contain only a few hundred cases of obfuscation, and our dynamic analysis lets us gain insight into the de-obfuscated strings. Out of the twelve static-only features that we validated, only the detection of HTML tag attributes other than `src` had any false positives (11 %), but we do not consider it an exploitation technique and only use it to complement our analysis. While such shortcomings are unacceptable for general-purpose attack detection, in our case we are only concerned with post-hoc analysis of two datasets, and we comprehensively validate our method.

4 Analysis

To provide some context, we start our analysis with an overview of the two exploit databases XSSED and OPENBUGBOUNTY. We report all results relative to the 133.9 k exploits (82.9% of raw submissions) for which our analysis framework successfully extracted and combined static and dynamic data. The XSSED data contains 36.2 k exploits for 28.7 k websites

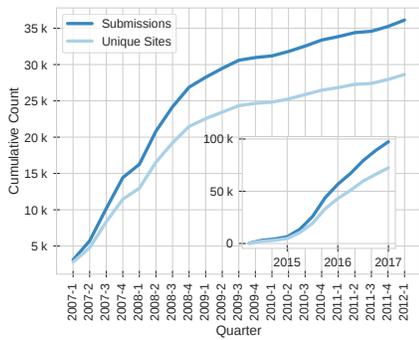


Figure 2: Quarterly exploit submissions and unique affected domains in XSSED (outer) and OPENBUGBOUNTY (inset). Most new submissions are found on new domains, suggesting a large supply of vulnerable domains on the Web.

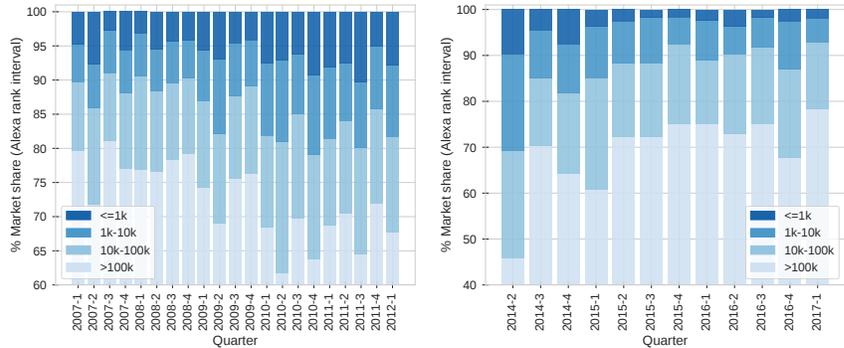


Figure 3: Quarterly distribution of the popularity of domains affected by exploit submissions (XSSed left, OPENBUGBOUNTY right). Domains grouped by popularity according to their Alexa ranks; the last interval includes unranked domains. More than half of submissions are for unpopular websites. The rank interval distribution is almost uniform over time, illustrating that XSS vulnerabilities continue to be found even on the most popular websites.

submitted between January 2007 and March 2015, although the site appears to have been mostly dormant since May 2012, as there have been fewer than ten monthly submissions since then. Our OPENBUGBOUNTY dataset covers June 2014 to April 2017 with 97.7 k submissions for 72.6 k websites. (We cannot practically extract newer submissions from OPENBUGBOUNTY due to newly implemented anti-crawling measures.)

The exploits in XSSED were submitted by 2.2 k users, whereas the much larger OPENBUGBOUNTY had only 883 active authors and 244 anonymous submissions. The dataset is heavily biased towards highly active users. The top 10 users accounted for 28.6 % of submissions in XSSED, and 38.3 % in OPENBUGBOUNTY. In contrast, around half of users in XSSED, and a third of users in OPENBUGBOUNTY, submitted only a single exploit. The dominance by a few active users implies that decisions made by these users, including their choice of exploits and their period of activity, can skew averages in our data. To account for this effect, we analyse aggregates not only in terms of submission quantities, but also by how many distinct authors submitted such exploits.

If submitters use automated tools, which appears to be the case for at least the most prolific users, there is a likely bias towards more shallow vulnerabilities, depending on the capabilities of these tools, and the possibility that submitted exploits contain techniques that are not necessary for the exploit to work on the respective website. Consequently, our analysis of exploits and their sophistication is to be read primarily as trends among exploit submitters, and as only loose upper bounds on the complexity of the website’s vulnerability.

4.1 Affected Websites

The submitted exploits refer to 28,671 unique registered domains (below the public suffix) in XSSED and 72,607 in

OPENBUGBOUNTY. Most domains receive a single submission; only 10.1 % and 13.3 % of them, respectively, appear in multiple vulnerability entries. These are often submitted in batches, as 37.2 % of delays between consecutive submissions for the same domain are below one day in XSSED, and 43.7 % in OPENBUGBOUNTY. Figure 2 shows that over time, total exploit submissions in both databases increase only slightly faster than the unique number of websites, suggesting that there is an “endless” supply of new websites with XSS vulnerabilities. In aggregate, the users submitting exploits appear to favour breadth over depth. The dataset is thus unsuitable for analysing the security posture of individual websites.

Since exploit authors are free in their choice of websites to scrutinise for XSS vulnerabilities, and we do not know about unsuccessful attempts, our data does not allow conclusions about vulnerability rates of websites. In the following, we study vulnerability reports according to the popularity of the affected website. We utilise the websites’ (presumably contemporaneous) Alexa ranks reported in the submissions, and group them into exponentially increasing intervals. More popular sites tend to be overrepresented in our data, as users submitted vulnerabilities for 38 % of the 100 most popular websites in XSSED (40 % in OPENBUGBOUNTY), but only 3.2 % (8.9 %) of the 90 k websites ranked in the range 10 k – 100 k according to Alexa. As shown in Figure 3, these proportions remain somewhat stable over time. This suggests that the ability of exploit authors to find XSS vulnerabilities even in the most popular websites has not changed significantly.

4.2 Sink Analysis

Injected exploits can be reflected by the website in different parts of the server response. Since reflection vulnerabilities often arise from server-side templating with

user inputs, such reflection sinks could for example appear between HTML tags `<div>{sink}</div>`, in an HTML attribute `<div name="{sink}"`, or in a JavaScript context `<script>var name="{sink}";`. We manually labelled two random samples of 250 submissions from each of the two databases to determine the sink context of the exploit reflection. Nearly all sinks appear in an HTML-related context. In both databases, sinks are located between HTML tags, or inside HTML tag attribute values at comparable rates, with 52 % and 46.4 % in XSSSED, and 44.4 % and 47.2 % in OPENBUGBOUNTY, respectively. JavaScript sinks make up only 1.2 % in XSSSED and 7.2 % in OPENBUGBOUNTY.

Depending on the sink context, the exploit may need to *escape* from the current context and set up a new context suitable for the payload [14]. If the payload is `alert(1)` and the sink is located inside the value of an HTML attribute such as `<div name="{sink}"`, the payload needs to terminate the string, and create an event handler attribute that can accept the JavaScript payload, e.g., `onmouseover=alert(1)`. Alternatively, it can close the tag, and then inject a script tag with the payload, such as `><script>alert(1)</script>`. When exploit authors manually craft their attack string, they can customise it using their understanding of the website. For example, only a restricted set of tags can trigger the `onload` event handler, whereas it would have no effect in the other tags. Alternatively, exploit authors can use generic escape sequences that work in a variety of different sink types in order to make their exploit more versatile. In the same two random samples, 50.8 % of exploits in XSSSED and 39.6 % in OPENBUGBOUNTY did not need any escaping, as the sink already had a JavaScript or HTML between-tag context suitable for the payload. However, 26.4 % of sampled XSSSED exploits (OPENBUGBOUNTY: 28 %) contain an escaping sequence even though it is not necessary. This suggests that our datasets contain a significant fraction of general-purpose exploits. Around 41.6 % of sampled XSSSED exploits, and 50.4 % in OPENBUGBOUNTY, contain an escape sequence that is both necessary and minimal. The remainder contains either no escaping, or additional unnecessary escaping.

Many websites reflect injected exploits more than once. In XSSSED and OPENBUGBOUNTY, 37.0 % and 32.5 % of submissions in our successfully merged dataset have more than one working exploit reflection. This typically occurs due to one URL parameter being used in multiple places in the server-side template, but there are also a few submissions where different exploits are injected into multiple parameters. In the manually labelled sample, the multiple working reflections occur in 45.2 % for XSSSED and 30.0 % for OPENBUGBOUNTY. These can be further divided into 32.4 % of the XSSSED sample, and 24.4 % of OPENBUGBOUNTY, where the exploit is reflected multiple times in sink contexts of the same type, and 12.8 % and 5.6 %, respectively, with multiple reflections in different sink contexts. An exploit with correct escaping for one context can also appear in a different context

where the escaping sequence may be ineffective.

To that end, it is worth noting that 44.7 % of all XSSSED submissions, and 52.9 % of OPENBUGBOUNTY contain at least one additional potential exploit reflection where the exploit does not execute. This data is to be seen as a coarse approximation, as it contains false positives that are not actual exploit reflections, but matches between the request data and similar but potentially unrelated page code. For this reason, outside of this section, we only analyse executing reflections, where our dynamic analysis rules out such false positives. Overall, 62.1 % of submissions in XSSSED, and 63.4 % in OPENBUGBOUNTY, contain multiple reflections, with at least one working and potentially more that do not. In addition to exploits being reflected in a sink context for which the escaping sequence is ineffective, another possible explanation for reflections not executing are server-side transformations.

A typical server-side transformation is encoding of sensitive characters to prevent XSS. In exploit reflections that do execute, few special characters such as `<` or `>` for tags, and `"` or `'` for attribute values or strings are HTML entity encoded (i.e., `<`; `<`; or `<`) – less than 0.1 % for angled brackets, and no more than 0.25 % for quotes in either database. In reflections that do not execute, the share of such encoding is significantly higher, with 7.3 % of these reflections in XSSSED containing HTML-encoded angled brackets (OBB: 8.7 %), and 6.0 % (8.0 %) containing encoded quotes. HTML-encoding of alphanumeric characters is close to zero in either case. Since many server responses contain both working and non-working reflections, vulnerable applications appear to sanitise some user input, but inconsistently.

Some reflections appear to be mirroring the full request URL instead of a single request parameter. Around 6.4 % of submissions in XSSSED and 15.1 % in OPENBUGBOUNTY contain at least one URL reflection, and 2.7 % and 4.4 % of submissions, respectively, contain at least one such URL reflection that executes. Similar to HTML encoding, URL reflections that execute injected exploits rarely contain any URL-encoded characters at all (XSSSED: 3.6 %, OPENBUGBOUNTY: 0.9 %), whereas non-executing URL reflections do (81.5 % in XSSSED and 73.0 % in OPENBUGBOUNTY).

Other factors that might prevent execution of an injected exploit, which we do not examine here, include an incompatible sink context, other types of encoding or escaping, and more complex server transformations such as substrings.

4.3 Exploit Analysis

Nearly all exploit submissions contain simple proof-of-concept payloads showing a JavaScript dialogue with a message. In XSSSED, the earlier dataset covering five years since 2007, 99.7 % of all submissions use `alert()`. However, the prevalence of `alert()` appears to be decreasing over time in favour of `prompt()`. The former is used in 52.4 % of OPENBUGBOUNTY submissions, the latter in 40.7 %, and 7.6 %

use `confirm()`. These numbers may be heavily dependent on the behaviour of a few very active users, as 86.1 % of all authors in OPENBUGBOUNTY submit at least one exploit with `alert()`, and the fraction of active authors with at least one such exploit remains over 77.4 % in each quarter.

4.3.1 Overview of Exploitation Techniques

We group exploitation techniques into five categories, as shown in Table 4: Context (Escaping and Creation), Syntax Confusion, String Obfuscation & Quote Avoidance, Control Flow Modification, and String Interpretation. Each category corresponds to a specific goal of exploit authors, such as bypassing (incomplete) server-side sanitisation, or setting up a context where JavaScript code can be executed. The techniques within each category are alternative means of achieving that goal. Exploits might use multiple exploitation techniques at once. Very few submissions (17.3 % in XSSED, 3.8 % in OPENBUGBOUNTY) use no special technique at all; they are simple exploits such as `<script>alert(1)</script>`. The most common category is context escaping and creation with 75.5 % of submissions in XSSED, and 83.8 % in OPENBUGBOUNTY, most likely because techniques of this category are often needed to set up the proper execution context for the exploit, depending on the sink type. The remaining categories appear in only a small fraction of exploits in the early XSSED, but become more popular in the later OPENBUGBOUNTY. As an illustration, 67.6 % of submissions in XSSED use no technique other than possible context escaping and creation, showing that older exploits tend to be relatively simple. In OPENBUGBOUNTY, this percentage is only 15.6 % of submissions, as some techniques gained popularity and more authors have submitted at least one exploit with a technique from the other categories.

While categories such as control flow modification and syntax confusion have an upwards trend in OPENBUGBOUNTY, both in terms of submissions and authors, the string interpretation category remains rare, appearing in less than 1 % of submissions in either database, and used by only 1.9 % of XSSED and 5.0 % of OPENBUGBOUNTY authors. In the following, we look into each category in more detail.

4.3.2 Context (Escaping and Creation)

Depending on the sink context (Section 4.2), exploits need to escape from the current context and set up their own in which the payload can run. In our two datasets, most exploits (73.8 % in XSSED, and 77.6 % in OPENBUGBOUNTY) close the previous tag, escaping to a context where new HTML tags can be inserted (e.g., `><script>`, C3 in Table 4). Only a few exploits escape from an HTML attribute but remain inside the tag to insert an event handler, such as `" onload=` (C4 in Table 4, 1.1 % in XSSED and 3.1 % in OPENBUGBOUNTY). Interestingly, though, a much higher fraction of

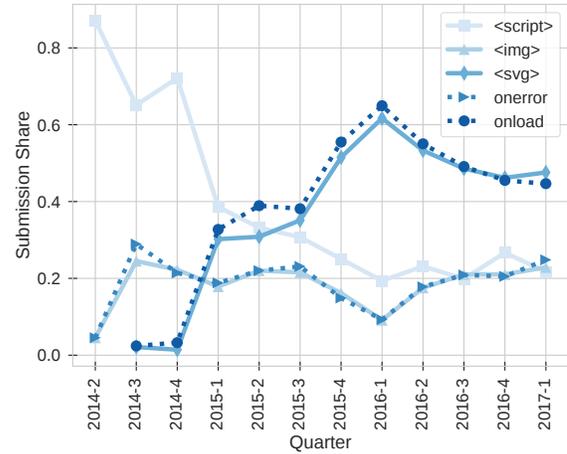


Figure 4: Quarterly tag and event handler market share (in terms of exploit submissions) in OPENBUGBOUNTY. Script tags decline in favour of alternative tags with event handlers.

authors, 20.4 % in OPENBUGBOUNTY, have submitted one such exploit at least once. Even fewer exploits insert their dialogue-based payload directly into a JavaScript context by possibly terminating a string and chaining the statement with `;` or an operator such as `+` (C5 in Table 4, 0.9 % in XSSED and 1.8 % in OPENBUGBOUNTY). While the latter finding is probably due to JavaScript sinks being much less common in our datasets than HTML sinks, the dominance of HTML tag escaping over remaining inside the tag is more likely attributed to preferences of exploit authors, as both types of HTML sinks are similarly common.

As most exploits close the previous tag, they must insert a new tag in order to be able to execute the payload. Indeed, 98 % of exploits in XSSED, and 93.5 % in OPENBUGBOUNTY contain at least one tag. In the older XSSED dataset, with 95.6 % of submissions, this is nearly always a `<script>` tag. In the more recent OPENBUGBOUNTY, this tag is found in an average of only 26.8 % of submissions. As Figure 4 shows, its use declined from 87.1 % of submissions in the second quarter of 2014 to 21.7 % in the first quarter of 2017. Author numbers similarly declined from 100 % to 65 % submitting at least one exploit with a `<script>` tag in a quarter. While barely used at all in the beginning of OPENBUGBOUNTY, `<svg>` and `` tags have become more popular, with an overall use in 45.6 % and 18.5 % of exploits, respectively. Interestingly, many of these `<svg>` submissions appear to originate from a smaller set of users, as only 34.8 % of authors have ever submitted such an exploit. In the case of ``, the trend is opposite. The fewer submissions were made by a larger 49.3 % of authors. Presumably, alternative tag names can circumvent filtering in websites. They appear to be used in combination with event handlers, which we investigate in Section 4.3.5.

Table 4: Detected Exploitation Techniques by Category.

(● static, ○ dynamic, ● combined detection methodology; percentages for XSSSED / OPENBUGBOUNTY)

Context (Escaping and Creation)					
Technique	Example	Detection	Submissions	Authors	
C1	HTML comment	--><script>alert(1)</script>	●	2.4% / 10.5%	9.2% / 21.7%
C2	JavaScript comment	/** */prompt(1)//	○	0.5% / 3.3%	2.7% / 12.9%
C3	HTML tag escape and tag insertion	"><script>alert(1)</script>	●	73.8% / 77.6%	66.7% / 80.7%
C4	HTML attribute escape and event handler	" autofocus onfocus=alert(1)	○	1.1% / 3.1%	5.6% / 20.4%
C5	chaining onto prior JavaScript expression	"-alert(1) or ;prompt(1)	○	0.9% / 1.8%	5.0% / 14.4%
<i>(any in category)</i>				75.5% / 83.8%	68.6% / 83.6%
Syntax Confusion					
Technique	Example	Detection	Submissions	Authors	
S1	extraneous parentheses	(alert)(1)	○	0.0% / 0.3%	0.0% / 1.8%
S2	mixed case	<scRipT>alert(1)</scRipT>	○	4.9% / 4.4%	8.8% / 19.8%
S3	JavaScript encoding (uni, hex, oct)	\u0061lert(1) or top["\x61lert"](1)	○	0.0% / 0.1%	0.1% / 2.3%
S4	malformed img tag	<script>alert(1)</script>">	○	0.2% / 0.2%	0.6% / 1.6%
S5	whitespace characters	<svg%0Aonload%20=_alert(1)>	○	0.0% / 0.0%	0.1% / 0.7%
S6	slash separator instead of space	<body/onpageshow=alert(1)>	○	0.1% / 42.8%	0.1% / 31.7%
S7	multiple brackets (tag parsing confusion)	<<script>alert(1)<</script>	○	8.2% / 1.7%	5.8% / 8.8%
<i>(any in category)</i>				13.0% / 47.8%	14.2% / 39.7%
String Obfuscation & Quote Avoidance					
Technique	Example	Detection	Submissions	Authors	
O1	character code to string	alert(String.fromCharCode(88,83,83))	●	4.5% / 3.7%	11.1% / 11.7%
O2	regular expression literal	prompt(/XSS/)	●	13.7% / 65.1%	16.7% / 62.8%
O3	base64 encoding	alert(atob("WFNT"))	●	0.0% / 0.1%	0.0% / 0.6%
O4	backtick	prompt`XSS`	○	0.0% / 2.4%	0.1% / 8.3%
<i>(any in category)</i>				18.2% / 71.1%	25.0% / 67.7%
Control Flow Modification					
Technique	Example	Detection	Submissions	Authors	
F1	automatically triggered events	<svg onload=alert(1)>	●	1.2% / 48.2%	5.9% / 38.1%
F2	exploit-triggered events	<input autofocus onfocus=alert(1)>	●	1.2% / 20.8%	3.6% / 50.6%
F3	user interaction events	onmouseover=prompt(1)	●	1.0% / 1.7%	5.1% / 16.9%
F4	dialogue assignment to variable	a=alert;a(1)	●	0.6% / 0.1%	0.7% / 2.3%
F5	throw exception	window.onerror=alert;throw/1/	●	0.0% / 0.2%	0.0% / 1.6%
F6	page code interaction	exploit invoked by existing code in the page (call stack)	○	0.0% / 0.1%	0.3% / 4.2%
<i>(any in category)</i>				4.1% / 70.8%	11.3% / 67.7%
String Interpretation					
Technique	Example	Detection	Submissions	Authors	
I1	document.write	document.write("<script>alert(1)</script>")	●	0.1% / 0.1%	0.8% / 0.5%
I2	eval	eval("alert(1)")	●	0.7% / 0.2%	1.3% / 2.9%
I3	setInterval/setTimeout	setTimeout("alert(1)", 20000)	●	0.0% / 0.0%	0.0% / 0.2%
I4	top/window key access	top["alert"](1)	○	0.0% / 0.1%	0.0% / 1.5%
<i>(any in category)</i>				0.8% / 0.4%	1.7% / 4.5%
Summary					
Technique	Example		Submissions	Authors	
<i>(no technique used at all)</i>		<script>alert(1)</script>		17.3% / 3.8%	50.7% / 42.2%
<i>(no technique or Context Creation)</i>		"><script>alert(1)</script>		67.6% / 15.6%	84.0% / 61.4%

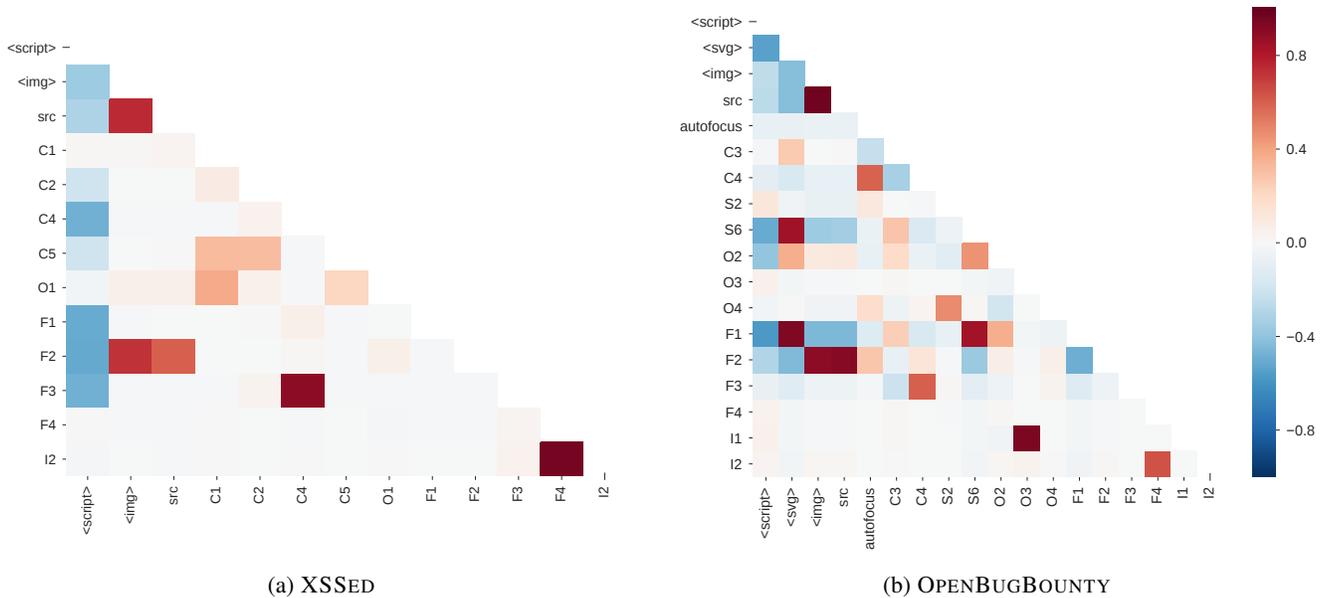


Figure 5: Correlation matrix of exploitation techniques and selected tags and attributes for (a) XSS and (b) OpenBugBounty. Use of `<script>` is negatively correlated with most other techniques, meaning they are used alternatively. The popular exploit pattern `` is visible as the pairwise correlation between ``, `src`, and `F2`. Rarely used techniques such as `I1` and `O3`, or `I2` and `F4`, may be correlated because most uses occur in similar submissions by the same author.

4.3.3 Syntax Confusion

Under Syntax Confusion, we consider techniques using non-standard or intentionally “confusing” syntax that is tolerated by browsers, but not understood by web application filters and their parsers. Only three techniques of this category see non-negligible use in the two exploit databases. In OpenBugBounty, 42.8% of submissions use a slash instead of a space (`S6` in Table 4; `<svg/onload>`); this is mostly due to its use in a frequently submitted exploit pattern (Section 4.4). The slash technique is barely used in the older XSS. On the other hand, using multiple angled brackets in an apparent attempt to confuse simple filter parsers (`S7` in Table 4; `<<script>`) is most popular in XSS with 8.2% of submissions. Both databases see a low but persistent use of mixed case tag names or event handlers (`S2` in Table 4; `<scRipT>`). Other syntax confusion techniques, such as JavaScript source code character encodings (`S3` in Table 4; `\u0061lert(1)`) or whitespace characters (`S5`; `<svg%0Aonload>`), appear in very few submissions.

4.3.4 String Obfuscation & Quote Avoidance

In this category, we group obfuscation using built-in JavaScript methods to decode character code sequences or base64 strings containing potentially filtered keywords, and use of regular expression literals or backticks to avoid injecting quotes and/or parentheses. The regular expression technique

(`O2` in Table 4; `alert(/message/)`) is used in a majority of OpenBugBounty submissions, but much less in XSS. The three remaining techniques occur in small percentages of exploits. Two of them, character code sequence decoding (`O1` in Table 4) and backticks (`O4` in Table 4; `alert`message``), are known to a larger fraction of authors, but used in fewer submissions overall.

4.3.5 Control Flow Modification

Control flow modification techniques mostly avoid straightforward use of potentially filtered `<script>` tags in favour of alternative ways of executing code. None of them is used widely in XSS, but automatically triggered event handlers (`F1` in Table 4) can be found in almost half (48.2%) of OpenBugBounty submissions. The most frequently used event handler of this category is `onload` (47.7% of OpenBugBounty submissions, 38.1% of authors). Other event handlers can be triggered automatically if the exploit contains the appropriate setup (`F2` in Table 4). Among those, we observe `onerror` together with an invalid URL (18.7% of submissions, 48.6% of authors), and `onfocus`, which commonly co-occurs with the `autofocus` attribute (2.1% of submissions, 12.2% of authors). Comparing the submission to author percentages, `onload` appears to be used by disproportionately active submitters, whereas `onerror` is known to more users, but submitted less frequently. OpenBugBounty exploits with

XSSed					
Rank	Techniques	Example	Submissions	Authors	
1	1	C3	"><script>alert(1)</script>	49.0 %	53.2 %
2	2	(none)	<script>alert(1)</script>	17.3 %	50.7 %
3	3	C3, O2	"><script>alert(/XSS/)</script>	7.8 %	11.0 %
4		C3, S7	"><script>alert(1)</script>>	7.3 %	3.8 %
5	4	O2	<script>alert(/XSS/)</script>	3.4 %	8.0 %
	5	C3, S2	"><ScRipt>alert(1)</sCripT>	2.9 %	5.4 %
OPENBUGBOUNTY					
Rank	Techniques	Example	Submissions	Authors	
1		C3, S6, F1, O2	"><svg/onload=prompt(/XSS/)>	30.9 %	18.5 %
2	3	C3, F2, O2	">	9.6 %	27.7 %
3	1	C3	"><script>alert(1)</script>	8.5 %	42.1 %
4	4	C3, O2	"><script>alert(/XSS/)</script>	4.7 %	26.0 %
5	2	(none)	<script>alert(1)</script>	3.8 %	42.1 %
	5	F2, O2		2.7 %	23.4 %

Table 5: Most common exploit patterns (ordered by submissions; rank for author use in second column).

event handlers that require user interaction (F3 in Table 4) are dominated by `onmouseover` (1.5 % of submissions, 14.5 % of authors) and `onmousemove` (0.1 % of submissions, 2.0 % of authors).

The most frequent combinations of tags and event handlers in OPENBUGBOUNTY are `<svg>` with `onload` (45.1 % of submissions), `` with `onerror` (18.0 %), `<iframe>` with `onload` (1.2 %), `<body>` with `onload` (1.1 %), and many additional combinations each used in fewer than 1 % of submissions. As shown in Figure 4, event handlers in OPENBUGBOUNTY have not always been equally popular. Rather, their rise coincides with a decline in the use of `<script>` tags, which are more likely to be filtered than the more innocuous-looking `<svg>` or `` tags. Direct code execution with `<script>` tags appears to be replaced by indirect execution using a combination of alternative tags and event handlers.

We also detect technically interesting techniques, such as interleaving exploit code so that it will be called by existing page code (F6 in Table 4), but they are observed rarely.

4.3.6 String Interpretation

Code and markup string interpretation methods such as `eval()` and `document.write()` could be used for custom exploit obfuscation. Our analysis of the 491 strings passed to these methods revealed only 9 additional instances of techniques that were not already visible during the static analysis. In the two databases, string interpretation methods appear to be used as proof-of-concept exploits rather than for actual obfuscation. Overall, the use of string interpretation techniques is very low, both in terms of submissions and authors. The most frequent technique is `eval()` (I2 in Table 4) with 0.7 % of submissions in XSSed, and 0.2 % in OPENBUGBOUNTY (1.3 % and 2.9 % of authors, respectively).

4.4 Exploit Patterns

Many exploitation techniques co-occur in characteristic combinations. (A correlation matrix is shown in Figure 5.) This suggests that authors may be using common patterns or templates for exploits. An interesting question is whether users submit *similar* exploits. To this end, we do not consider string equality to be a useful metric, as two exploits displaying different messages may be identical from a technical point of view, even though their string representation differs. Instead, we cluster submissions by the set of techniques (in Table 4) that are used in the exploit. We do not distinguish whether exploits use `alert()` or `prompt()`, do not consider tag names, and we only make a difference between three classes of event handlers. This level of abstraction is intended to balance robustness against minor modifications that do not result in a different control flow, yet reflect the author's use of exploitation techniques such as syntax tricks.

Our approach results in 178 distinct exploit patterns in XSSed, out of which 60.1 % are submitted at least twice and 51.7 % are used by at least two authors. In OPENBUGBOUNTY, we detect 484 exploit patterns, with 65.9 % used multiple times and 54.3 % used by multiple authors. For comparison, exact string matching finds 15,567 and 31,337 unique exploit strings, with only 13.5 % and 12.6 % of them used more than once (2.9 % and 3.4 % used by multiple authors).

The ten most frequent patterns account for 92 % of all submissions in XSSed, and 69.9 % in OPENBUGBOUNTY. Out of all authors, 93.8 % in XSSed, and 85.4 % in OPENBUGBOUNTY, have submitted at least one exploit based on one of these top ten patterns. Example string representations of these patterns are shown in Table 5. An example for the most frequently submitted pattern in XSSed is `"><script>alert("XSS")</script>`; it accounts for 49.0 % of all submissions and is used by 53.2 % of users. The

same pattern, or its variant without tag closing, is also the most popular in OPENBUGBOUNTY when considering the number of authors who have submitted it at least once. However, both patterns together account for only 12.3 % of total submissions. The most frequently submitted OPENBUGBOUNTY pattern is "<svg/onload=prompt (/XSS/)>", adding space and quote avoidance, and indirect code execution using an automatically triggered event handler. It accounts for 31.3 % of all submissions, but is used by only 19.1 % of users, which implies that these participants are disproportionately active.

4.5 Findings and Implications

We summarise below the main findings and implications of our analysis.

- Roughly half of submissions in both databases contain at least one non-executable input reflection in addition to the working exploit. Web developers may be aware of the need for input sanitisation, but apply it inconsistently.
- The older XSSED contains 67.6 % submissions with no particular exploitation technique at all or just context escaping, whereas almost half in the newer OPENBUGBOUNTY use automatically triggering event handlers. This trend away from direct execution using injected script tags towards indirect execution using non-script tags is presumably to circumvent simple input filters.
- Some techniques, such as using a slash instead of a space for separation <svg/onload, or using a regular expression instead of quotes alert (/message/), are frequent in large part because a few submitters use them in their large bulk submissions. Especially OPENBUGBOUNTY is dominated by a small number of very active users, likely due to automation. The choices made by these users have an outsized effect on aggregate results when it comes to the prevalence of exploitation techniques.
- Many vulnerabilities appear to be detected using automated tools, as evidenced by large bulk submissions and general-purpose exploits that include unnecessary context escaping. If website operators proactively used similar tools to test their own websites, they might be able to prevent a large number of the submitted vulnerabilities.
- Users appear to favour breadth by covering new websites more than existing websites, but they are still able to find new vulnerabilities even on the most popular websites. The discovery of cross-site scripting vulnerabilities on the Web seems to be far away from saturation.
- The databases contain few submissions with more complex techniques such as control flow modification or string interpretation. Possibly due to a lack of incentives, users may be looking for new websites that are vulnerable to an existing general-purpose exploit, as opposed to searching for new exploits on an existing website.

5 Discussion & Conclusion

Our longitudinal analysis of exploitation techniques in XSSED and OPENBUGBOUNTY submissions has shown that most reflected server XSS exploits are surprisingly simple, with an only moderate increase in sophistication over ten years. For example, few exploits use obfuscation, possibly because users lack incentives to submit more complex exploits. Similarly, in additional experiments we found that the vast majority of exploits could have been blocked by the default settings of filters that were available in browsers at the time the exploits were submitted, such as XSS Auditor in Chrome, the XSS filter in Internet Explorer, or NoScript for Firefox.

Currently, submitters gain recognition primarily through the number of exploits they submit. This appears to encourage submitters to use automated tools and scan a nearly boundless supply of vulnerable websites with simple, general-purpose exploits. We believe that XSS databases could increase their utility by leveraging human skill for tasks that are more difficult to automate. Bug bounty programmes could encourage depth over breadth by restricting the set of eligible websites, or by scoring submissions by the “complexity” or uniqueness of the techniques needed to make the exploit work.

The relative simplicity of exploits in the two databases, also anecdotally observed by Pelizzi and Sekar [17], has implications for researchers using them for model training or system evaluation. Ideally, a sample of exploits used for these purposes should cover a diverse set of technical conditions. In reality, however, most exploits in the two databases are technically equivalent. As a result, random samples of exploits contain only a few complex examples that would challenge the system to be evaluated. For a more diverse sample, researchers could apply an approach similar to ours and select exploits based on different patterns.

Our data does not allow conclusions about the security posture of individual websites and how it evolves over time. Yet, standard, low-sophistication exploits appear to be effective on a large set of websites including the most popular ones, demonstrating that shallow XSS vulnerabilities are still extremely widespread on the Web.

We hope that our research will spark new directions with the goal of improving the security posture of existing websites and mitigating the prevalent XSS attacks. The data used in this paper is available online [1].

6 Acknowledgements

We would like to thank Ahmet Talha Ozcan for his help with the data collection. We thank our anonymous reviewers for their valuable feedback on drafts of this paper. This work was supported by the National Science Foundation under grant CNS-1703454.

References

- [1] Annotated XSS dataset for this paper. http://cdn.buyukkayhan.com/public/xss_sqlite.db, 2020.
- [2] Rudolfo Assis. XSS cheat sheet. <https://brutelogic.com.br/blog/cheat-sheet/>, 2017.
- [3] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.
- [4] Khalil Bijjou. Web application firewall bypassing - How to defeat the blue team. In *OWASP Open Web Application Security Project*, 2015.
- [5] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A look into 30 years of malware development from a software metrics perspective. In *RAID*, 2016.
- [6] K. Fernandez and D. Pagkalos. XSSed | Cross site scripting (XSS) attacks information and archive. <http://xssed.com>.
- [7] Matthew Finifter, Devdatta Akhawe, and David Wagner. An empirical study of vulnerability rewards programs. In *Usenix Security*, 2013.
- [8] Mauro Gentile. Snuck payloads. <https://github.com/mauro-g/snuck/tree/master/payloads>, 2012.
- [9] HackerOne.com. Bug bounty – hacker powered security testing | HackerOne. <https://hackerone.com/>.
- [10] Robert Hansen, Adam Lange, and Mishra Dhira. OWASP XSS filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2017.
- [11] Mario Heiderich. HTML5 security cheatsheet. <https://html5sec.org/>, 2011.
- [12] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of Java exploitation. In *CCS*, 2016.
- [13] Vladimir Ivanov. Web application firewalls: Attacking detection logic mechanisms. In *BlackHat*, 2016.
- [14] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later - Large-scale detection of DOM-based XSS. In *ACM CCS*, 2013.
- [15] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting. In *NDSS*, 2018.
- [16] OpenBugBounty.org. Open Bug Bounty | Free bug bounty program & coordinated vulnerability disclosure. <https://openbugbounty.org>.
- [17] Riccardo Pelizzi and R Sekar. Protection, usability and improvements in reflected XSS filters. In *ASIACCS*, 2012.
- [18] Jukka Ruohonen and Luca Allodi. A bug bounty perspective on the disclosure of Web vulnerabilities. In *WEIS*, 2018.
- [19] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Quo vadis? A study of the evolution of input validation vulnerabilities in Web applications. In *Financial Crypto*, 2011.
- [20] Ben Stock, Stephan Pfister, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *ACM CCS*, 2015.
- [21] Mingyi Zhao, Jens Grossklags, and Peng Liu. An empirical study of Web vulnerability discovery ecosystems. In *CCS*, 2015.

Mininode: Reducing the Attack Surface of Node.js Applications

Igibek Koishybayev
North Carolina State University
ikoishy@ncsu.edu

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

Abstract

JavaScript has gained traction as a programming language that qualifies for both the client-side and the server-side logic of applications. A new ecosystem of server-side code written in JavaScript has been enabled by Node.js, the use of the V8 JavaScript engine and a collection of modules that provide various core functionality. Node.js comes with its package manager, called NPM, to handle the dependencies of modern applications, which allow developers to build Node.js applications with hundreds of dependencies on other modules.

In this paper, we present Mininode, a static analysis tool for Node.js applications that measures and removes unused code and dependencies. Our tool can be integrated into the building pipeline of Node.js applications to produce applications with significantly reduced attack surface. We analyzed 672k Node.js applications and reported the current state of code bloating in the server-side JavaScript ecosystem. We leverage a vulnerability database to identify 1,660 vulnerable packages that are loaded from 119,433 applications as dependencies. Mininode is capable of removing 2,861 of these vulnerable dependencies. The complex expressiveness and the dynamic nature of the JavaScript language does not always allow us to statically resolve the dependencies and usage of modules. To evaluate the correctness of our reduction, we run Mininode against 37k Node.js applications that have unit tests and reduce correctly 95.4% of packages. Mininode was able to restrict access to the built-in `fs` and `net` modules in 79.4% and 96.2% of the reduced applications respectively.

1 Introduction

Node.js [10] is an open-source JavaScript runtime engine typically used to build scalable network applications. The JavaScript runtime that powers Node.js is based on Chrome's V8 engine. Despite Node.js' young age, it has become very popular among the open-source community and enterprises. Moreover, big companies such as Microsoft, IBM, PayPal [22, 27, 39] are among others who use Node.js in their

products. One of the reasons for its popularity is in Node.js architecture choice. Node.js uses a non-blocking event-based architecture which gives an ability to developers to scale up Node.js applications easily. Nowadays Node.js is used to develop critical systems [49] that require security attention.

Node.js developers distribute community-developed libraries using an in-house built package manager system called NPM. NPM is considered to be the largest package manager by the number of packages [12] it hosts (over million) and growth rate of almost 800 pkg/day [9]. Since 2014, the NPM registry traffic has grown 23,500%, which shows its increasing popularity among developers [47]. This staggering amount of packages hosted in NPM gives developers the power to build apps very quickly by using already implemented functionality by others. In this paper, we argue that overusing third-party libraries comes with its own security risks.

The drawbacks of extensive dependence on third-party packages are: (1) developers need to trust others on the security and maintenance of the libraries; (2) the popularity of NPM makes it lucrative for adversarial users to distribute malicious libraries using attacks such as typosquatting [20, 43, 44], ownership takedown and introducing a backdoor [45, 52]; (3) upgrade or removal of the package from NPM may break the build pipeline of an application [46].

Our study of 1,055,131 packages shows that on average only 6.8% of the code in the application is original code according to source logical lines of code (LLOC) or putting in different words 93.2% of the code in Node.js application is developed by third-parties. One of the reasons why developers tend to use "trivial" third-party packages, is the belief that they are well managed and tested. Despite the belief, the study shows that only 45.2% of "trivial" packages have tests implemented [19].

Previous works on Node.js security mostly concentrate on architecture choice of Node.js and, therefore, on attacks that target the main thread of Node.js applications [23–25, 38, 42]. Others have conducted research on the reasons why developers use "trivial" dependencies [19] and security implications of depending on NPM packages [52]; however, no research

on an attack surface that extensive usage of third-party libraries may bring and ways of reducing the attack surface of a Node.js application was conducted before.

Due to all of the above, it is important to know the attack surface of third-party packages and to reduce them ahead of time during a development process.

Our main contributions are the following:

- We developed a system that reduces the attack surface of Node.js applications by removing unused code parts and modules from the dependency graph. The system can use one of two different reduction modes: (1) *coarse-grain*; (2) *fine-grain*; Our system is publicly available here: <https://kapravelos.com/projects/mininode>
- We analyzed 672,242 Node.js applications from the NPM repository and measured their attack surface. Our findings show that at least 119,433 (11.3%) of applications depend on vulnerable third-party packages. Also, on average only 9.5% of all LLOC is used inside analyzed packages.
- We created a custom build of Node.js that can restrict the access to the built-in modules using a whitelist generated by Mininode. Our evaluation experiment shows that Mininode successfully restricted the access to `fs` and `net` modules for 79.4% and 96.2% of packages, respectively.

2 Background

In this section, we describe the technical details of Node.js modules, how NPM works, and how Node.js resolves imported modules, both built-in and third-party dependencies. We use the term **module** to describe anything under the `node_modules` folder that can be loaded using the `require` function. A **package** is everything that is hosted on NPM, but not necessarily can be loaded by using `require` function, *e.g.* CSS files. In this paper, we refer to a package as a directory of files with a JavaScript entry point that can be loaded with the `require` function. We treat applications the same as packages, *i.e.* they both have a JavaScript entry point, from which Mininode can start its analysis.

2.1 Node.js module system

JavaScript has been traditionally the language of the browser. Web applications build their front-end logic in JavaScript by leveraging browser APIs. Node.js applications rely on a completely different ecosystem that is built to assess the needs of server-side applications. Instead of the DOM and other Web Platform APIs, Node.js relies on built-in modules that provide functionalities like networking and filesystem access. These modules are based on the CommonJS module system and only

recently we have seen experimental support for ECMAScript modules [11].

Node.js treats every JavaScript file as a CommonJS module. Node.js has built-in `require()` function to import both built-in and developer-created modules into code. The `require` function behaves differently depending on the type of the module requested. If the requested module is on the list of built-in modules, then it is returned directly from the modules written in C++. If the requested module is not part of the built-in modules, `require` will wrap the imported module with a function wrapper, as shown in Listing 1, before executing the code. This ensures that variables from the imported modules are not placed unintentionally in the global scope. Despite this, modules can declare variables and functions in the global scope, which poses a challenge in accurately determining the used APIs of the module (§5.2).

```
1 (function(exports, require, module,
  __filename, __dirname) {
2 // Module code lives in here
3 });
```

Listing 1: Function wrapper to execute module code

Every module that wants to provide some of its functionality to other modules can use the `exports` object. For example, in Listing 2 `b.js` exports two functions. However, `a.js` uses only function `foo()` after importing `b.js`. Thus, function `bar()` can be removed without impacting the behavior of the `a.js`. We discuss how we leverage this mechanism to restrict access to built-in modules Section 6.1.

```
1 //inside b.js
2 exports.foo = function foo() {}
3 function bar() {}
4 exports.bar = bar;
5 //inside a.js
6 var b = require("./b.js")
7 b.foo()
```

Listing 2: Example of CommonJS module and common ways to export the functionality

2.2 Node Package Manager (NPM)

Node.js comes with a built-in package manager called NPM, which hosts aside from JavaScript libraries also front-end CSS, JavaScript frameworks and command-line tools. In this paper we focus only on server-side Node.js packages that are distributed over NPM. Developers can install a package using the command `"npm install <name>@<version>"`, where `"<name>"` is the name of the package. By default, if version is missing, NPM will install the latest version of the package. If the package name is not given, NPM will look for the `package.json` file inside the current working directory and will install all packages listed as dependencies in the file. The `package.json` file also contains metadata about the Node.js application. These metadata can contain the main file of the application (*i.e.* entry point), the version number, a short de-

scription, a list of dependencies, and other information about the Node.js application. NPM installs dependencies transitively. For example: if package *A* depends on package *B*, and package *B* depends on package *C*, NPM will install packages *A*, *B*, *C*. The NPM's transitive installation of dependencies creates a serious problem of bloated code, as it makes it really hard for the developer to understand on how many packages the code depends. De Groef *et al.* states that some popular packages may in total depend on more than 200 packages [26].

3 Threat Model

Our threat model targets vulnerable Node.js applications that are susceptible to arbitrary code execution vulnerabilities. The main premise of this paper is that we can 1) reduce the capabilities of the attack by restricting the application's functionality and 2) eliminate further exploitation of the application that would elevate the attacker's capabilities by targeting vulnerabilities in unused dependencies.

We reduce the attack surface of applications by restricting the available built-in modules that can be loaded to the absolute necessary. This results in removing classes of capabilities from the application if they are not already used, like filesystem access or networking.

We mitigate chained exploitation by removing unused vulnerable modules and restricting built-in modules. Our assumption here is that the application is exploitable, but with certain restrictions, i.e., code can be injected, but without arbitrary execution due to unsafe regular expression checks. In that particular scenario, the attacker can take advantage of other existing vulnerable packages that are now reachable and gain full control of the application. The attacker can take advantage of the existing modules in two scenarios: 1) directly manipulating the input to the `require` function to load any modules, 2) indirectly manipulate the input to the `require` function to load any modules, except built-ins. Our system is capable of stopping further exploitation of the unused parts of the application, but it does not prevent the initial vulnerability that leads to partial code execution.

When the attacker can directly manipulate the `require` function and load additional built-in modules, Mininode restricts the access to unused built-in modules, even if they are used in one of the unused transitive dependencies. This significantly reduces the capabilities of the attack.

Listing 3 shows a theoretical motivating example of chained exploitation when the attacker can indirectly control the input to the `require` function. In the example, the attacker can inject malicious data to the `fs.linkSync` function (line 6), which is used to create a symbolic link, by manipulating the request data. For example, the attacker can replace the entry point of `header-parser` with a symbolic link to `unused.js` by manipulating `dst` and `src` fields. Therefore, next time when the attacker navigates to `"/exploit"` endpoint, Node.js will load the `unused.js` module instead of the

`header-parser` package, and the application passes data provided by the attacker to the `unused.js` (lines 11-12). Note that an attacker cannot manipulate symbolic links to load built-in modules because they are part of the Node.js binary. For this kind of chained exploitation, Mininode can remove unused packages from the application and restrict the attacker's ability to load modules that are not used by the application; thus, making the attack less effective. Some vulnerable packages though, like `fast-http` [4], `marscode` [2] and `marked-tree` [3] are directly exploitable by just loading their module, but not all vulnerabilities can be exploited via chained exploitation, as they can depend on additional constraints that might not be available to the attacker.

```
1 const fs = require('fs')
2 const express = require('express')
3 const app = express()
4
5 // some code parts omitted for brevity
6 app.get('/vulnerable', (req, res) => {
7   fs.linkSync(req.body.dest, req.body.src);
8   res.send('Hello World!')
9 });
10
11 app.get('/exploit', (req, res) => {
12   let parser = require('header-parser');
13   let result = parser(req.headers);
14   res.send(result);
15 });
```

Listing 3: Motivating vulnerable example of chained exploitation for loading unused packages

One of the great advantages of Mininode is that it restricts the attacker from using *any* unused module, including built-in modules, e.g. `fs`, even if it is used in a transitive dependency. When these modules are not used from the application, Mininode can have a significant impact on the attack.

4 Design Goals and Architecture Overview

4.1 Design Goals

There were two main design goals that we followed during the implementation of the Mininode.

Effectiveness. Mininode should reduce the attack surface of the Node.js application as much as possible. To achieve the *effectiveness* goal, we implemented two modes of reduction: (1) coarse-grain; (2) fine-grain; and added a built-in module restriction mechanism into Node.js. We provide Mininode's reduction effectiveness results later in the paper (§7.1 and 7.2).

Correctness. Mininode must remove only unused code parts, i.e. should not break the original behavior of the application. To validate that Mininode meets the *correctness* goal, we automatically verified the original behavior of 37,242 packages after reduction (§7.1).

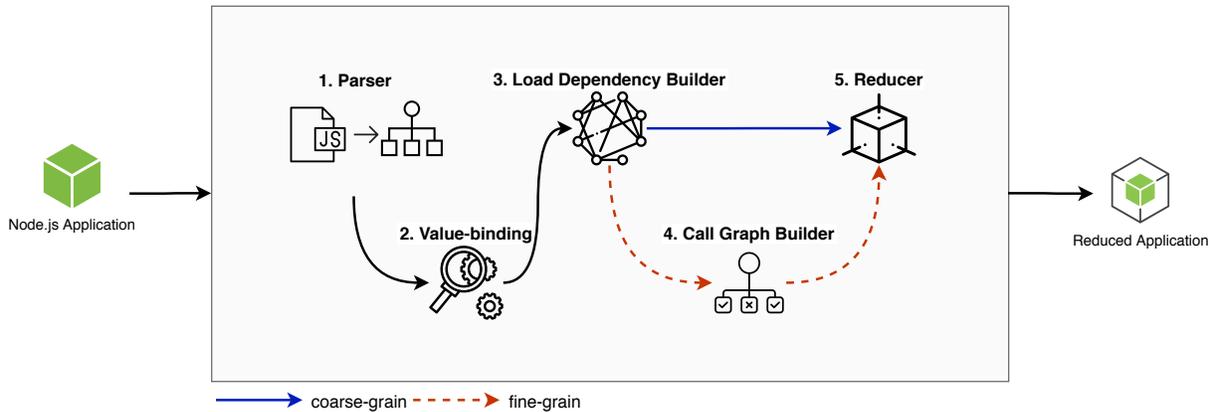


Figure 1: Mininode consists of main 5 parts: ❶ Parser; ❷ Value-binding; ❸ Load Dependency Builder; ❹ Call Graph Builder; ❺ Reducer and supports *coarse-grain* and *fine-grain* reductions

4.2 Architecture Overview

Mininode consists of five different stages as shown in Figure 1, that will run during the reduction of the application.

Mininode takes a directory path of a Node.js application, which contains *package.json* file, as an input. Then, Mininode traverses the directory, parses all JavaScript files and generates abstract syntax trees (ASTs) for each JavaScript file (❶). We use AST representation of the source code because it is easier to analyze statically and convenient to transform the tree structure to modify the initial source code. To parse JavaScript code, we use the popular open-source *esprima* [8] library. The final outcome of the *Parser* is an array of all modules inside the application folder and each module’s AST representation. After *Parser* completes processing the code, it will pass the generated ASTs to the next stages of Mininode.

The *Value-binding* (❷) is a pre-reduction stage that collects metadata about each module in the application from their AST representation, which is used in reduction stages. Also, *Value-binding* collects overall statistics on each module regarding the number of logical lines of code, dynamic imports, dynamic export.

The main reduction process consists of three different stages: (1) *Load Dependency Builder*, (2) *Call Graph Builder* and (3) *Reducer*.

The *Load Dependency Builder* (❸) builds a file-level dependency graph of the application by traversing the AST generated by *Parser*. To build the dependency graph, *Load Dependency Builder* starts from the entry point(s) and detects all *require* function calls from AST. All modules that are recursively accessible from the application’s entry point(s) are marked as used by *Load Dependency Builder*. Despite the simplicity of the algorithm, there are challenges that need to be addressed to construct a complete dependency graph (§5) and are further discussed in Section 6.

Mininode supports two reduction modes for the application: (1) *coarse-grain* reduction; (2) *fine-grain* reduction. The

coarse-grain reduction mode works at file-level and removes unused files in the application, while the *fine-grain* reduction works at function-level and removes functionalities from individual modules that are never used. As shown in Figure 1, Mininode skips the *Call Graph Builder* stage, and proceed directly to the *Reducer* stage in *coarse-grain* reduction mode.

The *Call Graph Builder* (❹) is responsible for detecting used and unused functions, exports and variables of all modules that are part of the dependency graph generated by *Load Dependency Builder* (§6.4). To achieve effectiveness design goal, *Call Graph Builder* may perform several passes on the AST of each module until no change to the final usage graph is made.

The final stage in our reduction pipeline is the *Reducer* (❺) stage. The *Reducer* is responsible for removing AST nodes and modules that are marked as unused by *Call Graph Builder* and *Load Dependency Builder*, respectively (§6.5). After finishing all of the reduction stages, Mininode generates source code for each module from their updated AST.

5 Challenges

The dynamic nature of JavaScript introduces several challenges to static analysis [33, 37, 50] and this is also true for the module system used by Node.js (§2.1). In this section, we list some of the research challenges that we faced during the implementation of the attack surface reduction tools using static analysis for Node.js applications. Overall, we divide the challenges into two categories: (1) export-related challenges; (2) import-related challenges.

5.1 Export-Related Challenges

The export-related challenges relate to the way how a module is exporting its functionality, and thus directly affects the Mininode’s *effectiveness* design goal as defined earlier (§4.1).

Failure to deal with export-related challenges will mostly lead to the *under*-reduction of the attack surface, *i.e.* not removing unused functionalities from the module. However, in some rare cases may lead to *over*-reduction of the functionality, *i.e.* removing used functionality. We give an example of both cases in follow up subsections.

Unusual use of the export object. JavaScript allows developers to modify an object in several different ways. Consequently, the export object can also be modified in several different ways to export functionality. For example, developers can create an alias for the export object and use the alias instead to export the module’s functionalities.

```

1 // inside request.js
2 exports.post = function() {}
3 exports.get = function() {}
4 // inside request-v2.js
5 exports = module.exports = require("request")
6 exports.patch = function() {}
7 // inside index.js
8 var req = require("request-v2")
9 req.get();
10 req.patch();

```

Listing 4: Example of re-exporting the imported module

Extension of a module with re-export. In CommonJS module system one can extend other modules by re-exporting it and adding additional functionality. In the example given in Listing 4, one can see how *request-v2.js* module extends *request.js* module with *patch* function. During analysis, Mininode should detect that *get* function used in *index.js* is actually coming from *request.js*. This behavior prevents *over*-reduction of the *request.js* module, and it ensures that Mininode meets the *correctness* goal.

5.2 Import-Related Challenges

The import-related challenges affect the static analysis’s performance in the detection of used functionalities of the imported CommonJS modules. Failure to detect used functionalities of imported modules will lead to *over*-reductions, *i.e.* removing used functionalities. Thus, it will lead to breaking the original behavior of the Node.js application. The rest of the section is describing some of the import-related challenges.

```

1 //inside request.js
2 exports.post = function() {}
3 exports.get = function() {}
4 // inside index.js
5 request = require("request");
6 request.post()
7 // inside util.js
8 request.get();

```

Listing 5: Example of importing a module in the global scope.

Dynamically importing the module. It is common for Node.js applications to load different modules depending on the execution environment or the user’s input. Dynamically

importing a module restricts the ability for simple static analysis to detect which module was loaded, which may lead to the removal of the whole used module. Therefore, it is vital to resolve dynamic imports to build a complete load dependency graph of the application.

```

1 // inside parent.js
2 module.exports = require("child.js")
3 exports.foo = function() {
4   exports.childFoo(); //defined in child.js
5 }
6 exports.parentBar = function() {
7 }
8 //inside child.js
9 exports.childFoo = function() {
10   exports.parentBar(); //defined in parent.js
11 }

```

Listing 6: Example of invisible parent-child dependency

Importing as a global variable. As discussed in Background section 2.1, Node.js wraps the modules with wrapper function to avoid collision of variable and function names and create separate scope for each module. Despite this, developers can import a module into a global scope, as shown in Listing 5. If a module is imported into a global scope, any other module can have direct access to the module’s functionality without importing it. Listing 5 gives an example of loading the *request.js* in global scope in *index.js*, which makes it possible for *util.js* to use *get* function of the *request.js* without importing it.

```

1 //in index.js entry point
2 var foo = require("foo")
3 var bar = require("bar")
4 foo.x()
5 bar.z()
6 //inside foo.js module
7 var bar = require("bar")
8 exports.x = function() {}
9 exports.y = function() {
10   bar.w()
11 }
12 //inside bar.js module
13 exports.w = function() {}
14 exports.z = function() {}

```

Listing 7: Example of cross-reference dependence.

Invisible parent-child dependency. This issue arises when the imported module (child) is using the functionality defined inside the module (parent) that imports the child module as shown in Listing 6. Because of the absence of a clear dependency link from child to parent, this challenge is counter-intuitive in nature. From Listing 6 one can see that, even if *child.js* is not importing *parent.js*, the child module is using *parentBar* that was defined in the parent module. We saw this behavior in one of the most popular NPM package *debug* [5].

Cross-reference dependency. The cross-reference dependency problem happens when two different modules import the same module, but they use different parts. For example

in Listing 7 *index.js* and *foo.js* are referencing *bar.js*, however using different parts of it. If Mininode preserves all used functionality, it will preserve `exports.w` function of *bar.js*, because function `exports.w` was used inside *foo.js*. However, function `exports.w` of *bar.js* should be removed because it is not reachable from the entry point (*index.js*) of the application.

6 Implementation

Mininode takes as input a Node.js application and makes three different reduction stages to produce a reduced version of the application (§4.2). This section gives implementation details of each reduction stage and how certain challenges described in Section 5 are resolved. Additionally, the following subsections give details about how access to built-in modules is restricted, and what kind of metadata is collected during *Value-binding* stage.

6.1 Restricting Access to Built-in Modules

The `require` function in Node.js checks if the requested module is in a built-in modules list before trying to resolve it (§2.1). We modified the original behavior of the `require` function at the Node.js C++ level. The patched `require` function restricts access to the built-in modules by checking if the requested module is not in a whitelist of built-in modules generated by Mininode. The whitelist is generated only once during the reduction of the application and kept unchanged. If the application does not have previously generated whitelist, our custom-built Node.js will allow all built-in modules to avoid breaking the application.

6.2 The *Value-binding* Details

The *Value-binding* (②) is a preprocessing step that collects metadata about each module in order to help other reduction stages to overcome challenges listed previously (§5). *Value-binding* collects an array of *aliases* for `exports` object and `require` function, because developers may rename these CommonJS APIs by assigning them to another variable and using an *alias* for the API instead of using API directly. Especially, this behavior can be seen in the case of packages that provide a minified version. Minification usually substitutes longer names with shorter ones to decrease the size of the file.

Value-binding also collects a dictionary of identifier names with their corresponding values, which are used to detect the possible values of dynamically imported modules. Possible values of identifiers could be literals (strings) or other identifier names that were assigned to the original identifier. If an identifier's value depends on any dynamic expression, e.g. a function call, *Value-binding* will mark the identifier as *non-resolvable*. However, if the identifier's value depends on binary expression, e.g. `var a=b+"-production"`,

Value-binding tries to resolve the possible values by getting the values of "b" from the dictionary and adding the "-production" to it. Note that variable "b" in the dictionary must be resolvable. Otherwise, *Value-binding* will mark the variable "a" as non-resolvable.

6.3 The *Load Dependency Builder* Details

The *Load Dependency Builder* (③) is responsible for building the file-level dependency graph by looking for `require` (or its *aliases* (§6.2)) function calls in AST and by resolving the function's argument to one of the existing modules (§4.2). The *Load Dependency Builder* resolves `require`'s argument using Node.js default resolution algorithm if the argument's type is literal. In other cases, it will use a simple algorithm to resolve dynamic import.

Resolving dynamic imports. To resolve the dynamic imports, the *Load Dependency Builder* uses the dictionary generated by *Value-binding* in the previous stage. If the argument's type is an identifier and the dictionary contains values for the identifier, *Load Dependency Builder* iterates through the possible values and resolves to possible modules in the application. This process is one of Mininode's advantages over other open-source NPM packages implemented to build the dependency tree of an application [13, 29, 31]. On the other hand, if the identifier does not exist in the dictionary, or the identifier is marked as non-resolvable, *Load Dependency Builder* will mark a module as using complicated dynamic import that can not be resolved reliably using only static technique. If the dependency graph of the application contains a module with complicated dynamic import, Mininode stops performing further analysis and exits because the application under analysis cannot be reduced reliably without breaking its original behavior.

6.4 The *Call Graph Builder* Details

The *Call Graph Builder* (④) runs only during fine-grain reduction mode, as can be depicted from Figure 1. The goal of the *Call Graph Builder* is to detect which parts of the code are used for each module and mark unused ones. To achieve the goal, it performs two separate tasks on the module's AST each time during analysis.

The first task, which is called *marking unused*, is responsible for marking exports, functions, and variables as unused if they are not used inside or outside of the module according to an array of used exports of the analyzed module.

The second task, which is called *usage detection*, is responsible for constructing the used exports array for each imported modules of the currently analyzed module. It achieves this by recording the variable names initialized by `require` (or the *aliases* (§6.2)) function calls and detecting all member expressions (i.e. property accesses) for all recorded variable names.

Resolving cross-reference challenge. To achieve the *effectiveness* design goal discussed in Section 4.1, Mininode needs to resolve the *cross-reference* challenge. The cross-reference challenge can be solved by always running the *marking unused* task before the *usage detection* task for each module’s AST. One benefit of running *marking unused* before *usage detection* is that during *usage detection*, we can skip functions that are marked as unused. For example, for the module *foo.js* from Listing 7, by first running *marking unused* task, we mark `exports.y` as unused. Thus, *usage detection* will skip traversing `exports.y` during analysis, and, therefore, `exports.w` will not be included in an array of used exported functions of the *bar.js* module.

Resolving extension of the module by re-exporting. The *Call Graph Builder* internally keeps track of used re-exported modules during an analysis of the AST. Later on, *Call Graph Builder* will add re-exported modules into the *descendants* array of the *currently* analyzed module. In the case of Listing 4, *request.js* module will be added into the *descendants* array of *request-v2.js* module. Additionally, *request-v2.js* becomes part of the *ancestors* array of *request.js* automatically. Later, when *Call Graph Builder* analyzes *request.js* module, it passes all used properties of *request.js*’s ancestors (*i.e.* *request-v2.js*) to the *marking unused* task as an extra argument. In this particular case, the extra array argument will include `get` and `patch` function names. Thus, exported `get` function of *request.js* will not be marked as unused during the *marking unused* task. Therefore, *Reducer* will not remove used exported functions, and eventually, Mininode will preserve the correctness design goal (§4.1).

Resolving invisible parent-child dependency. *Call Graph Builder* resolves the invisible parent-child dependency challenge almost the same way it resolves extension by re-export challenge. Because Mininode already has information about ancestors and descendants of the module, *Call Graph Builder* can pass as an extra argument all used exports of all ancestors and descendants of the module during the *marking unused* task. In the case of Listing 6, *Call Graph Builder* pass as an extra argument *child.js*’s used exports array, which contains `parentBar` function, into the *marking unused* task for *parent.js*.

Resolving importing as a global variable. To resolve the importing module as a global variable challenge, *Call Graph Builder* keeps track of leaked global variables during the *usage detection* task and stores used members, *i.e.* accessed properties, of a variable inside a dictionary of leaked global variables. In the example from Listing 5, *Call Graph Builder* creates an entry in the dictionary with a key `request` after analyzing *index.js*. The value of the entry is an array of used members, which contains `post` and `get` after analyzing *index.js* and *util.js*, respectively. Next, when *Call Graph Builder* performs the *marking unused* task for *request.js*, it passes the corresponding members’ array of the dictionary as an extra argument.

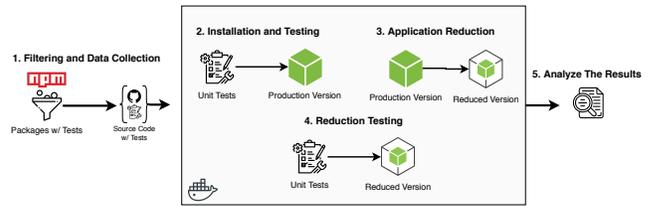


Figure 2: Validation experiment setup

6.5 The Reducer Details

The *Reducer* (5) is responsible for removing the AST nodes marked as unused without breaking the valid syntax of the AST and generating code from the AST. We are using the open-source *escodegen* [7] library to generate the source code from the AST.

Resolving unusual use of exports object Currently, *Reducer* can reduce exporting logic for the most common three ways to statically define a property for the object in JavaScript. These are: (1) defining property using dot notation, *e.g.* `exports.a=1`; (2) defining property using array notation, *e.g.* `exports["b"]=2`; and (3) defining property using `Object.defineProperty` function. In addition to the listed ones, *Reducer* tries to resolve the value for dynamically defined properties, *e.g.* `var c='c'; exports[c]=3`, using a similar algorithm as in the *resolving dynamic import challenge*. If the *Reducer* cannot resolve the dynamically defined property, it will not reduce the property, which may cause *under-reduction*. However, this behavior will not break the original code of the application.

7 Mininode Validation and Measurement

Overall, we run two experiments: (1) to validate the correctness of the Mininode in reducing the attack surface of the application; (2) to measure the bloated code and to check the effectiveness of the system in reducing the attack surface and vulnerabilities in the NPM registry packages. In the next subsections, we will give more details of experiments’ setup and results.

7.1 Mininode Reduction Validation

Experiment Setup. We performed the validation experiment to evaluate the *effectiveness* and *correctness* of the Mininode reduction (§4.1). We measured the effectiveness by calculating the total number of removed files, removed LLOC, and removed exports. The correctness of the reduction, *i.e.* whether Mininode reduced the package without changing its original behavior, is measured by the success rate of passed original unit tests of the packages after the reduction of their attack surface. The validation experiment consists of five steps as shown in Figure 2.

The goal of the first *Filter and Data Collection* step was to gather package names for which we could run unit tests, and calculate the tests coverage metrics automatically. The most popular test coverage package on NPM is *nyc*, previously known as *istanbul*, which is advertised as a tool where no configuration is needed to calculate unit tests' code coverage metrics. Therefore, we selected packages that list as one of their dependencies *nyc* and/or unit test package that is compatible with *nyc*. In total, 225,449 out of 1,055,131 packages depend on one of the packages required for automatic testing and coverage calculation. Next we collected packages' source code from Github, installed them, and ran their original unit tests without performing any reduction. We decided to collect source code from Github because not all developers publish package's test code into NPM. As a result of this, we were left with only 49,535 packages that were successfully installed, and passed their original unit tests before reduction.

Initially, we tried to reduce the full version of the packages and run the unit tests on the final results. However, this approach failed, because during the reduction step Mininode removed all code responsible for unit tests. To resolve this issue, we leveraged the way Node.js looks for a correct dependency by traversing the *node_modules* directory in the same location where the file requesting the dependency is located. We installed both the full and production versions of the package and created a symbolic link from the main file (*i.e.* entry point) of the full version to the main file of the production version. In this way, we could run the package's unit tests in the full version but test its production version. During the test of packages' production version, we noticed that some packages in production version require developer-only dependencies that are not installed (§2.2). Usually, these cases of counter-intuitive dependencies are used in the packages that are implemented as a plugin to other developer-only packages, *e.g.* *eslint-plugin-jest* is a plugin for *eslint*. Another challenge we faced was that *babel* [1], a popular package to transpile JavaScript, needed a special configuration for projects located in a different folder or symlinked [14]. After eliminating packages that failed during the test of their production version using a symbolic link from the full version, we were left with 45,045 packages in our validation dataset.

The final steps in the validation experiment, before result analysis, were package reduction and unit tests validation of the reduced version of the packages (Figure 2). In 6579 out of 45,045 packages Mininode detected dynamic import that could not be resolved with the current implementation (§6.3) and for 2.7% of packages Mininode threw runtime errors, such as heap out of memory. The final dataset has 37,242 packages that we tested for correctness and effectiveness.

Results. For the final dataset of 37,242 packages, we performed both coarse-grain and fine-grain reduction and ran unit tests to verify that Mininode did not break the original functionality of the reduced packages. The results of both modes of reduction are shown in Table 1. As it may be ex-

	Coarse-grained	Fine-grained
Passed test	35,762	35,531
Removed fs module	28,144	28,196
Removed net module	33,262	34,180
Removed http module	32,878	32,795
Removed https module	33,137	33,044
Total removed files	86.9%	87.3%
Total removed LLOC	85.4%	92.2%
Total removed exports	86.7%	89.0%
Failed test	1,480	1,711
TOTAL	37,242	37,242

Table 1: Coarse and fine grain reduction results on validation set

pected, the coarse-grain reduction (96.0%) has a higher success rate than the fine-grain reduction (95.4%). This is due to the behavior of the fine-grain reduction trying to reduce the individual modules on function-level, compared to coarse-grain reduction, which only performs reduction on file-level. Reducing in the fine-grain mode may cause *over-reduction* of the used functions, which leads to breaking the original behavior of the package and, thus, failing the unit tests. However, despite the higher failure rate, the fine-grain reduction performed better in terms of reducing unused code parts. Fine-grain reduction removed almost 8% more LLOC compared to coarse-grain reduction. Also, in other reduction categories such as reduction of the files and the exported functionalities, fine-grain shows better results. As shown in Table 1 Mininode was able to restrict the access to the built-in modules at least in 28,144 (78.7%) of packages during the coarse-grain reduction, and in 28,196 (79.4%) of packages during the fine-grain reduction. The high results of reduction may be counter-intuitive, especially in case of reducing a lot of files and LLOC from the package. That is why we randomly selected three packages that have a more than 99% reduction rate and manually verified the results. Both of the packages *mfdc-router* and *middleware-chain-js* were shipping a bundled version along with their source code. In these cases, Mininode removed almost all of their dependencies from the *node_modules* folder and unnecessary source code files. In the last case, after installation, *cpr* had 35,911 test files out of all 35,982 JavaScript files, which were removed by Mininode.

Coverage	Coarse-grained	Fine-grained
100%	13,561	13,548
Between 90-99.9%	8,413	8,290
Between 50-90%	6,915	6,797
Unknown or below 50%	6,873	6,896
Total	35,762	35,531

Table 2: Coverage statistics of successfully passed test samples

In addition, we calculated the test coverage of the success-

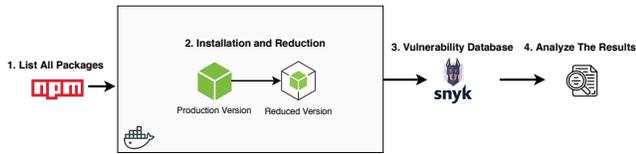


Figure 3: NPM measurement experiment setup

Job statuses and reasons	Packages
Succeeded packages	672,242
Failed packages	382,889
Package does not have main entry point	188,630
Non-resolvable dynamic import detected	128,533
Failed to install	26,875
Package's main entry point is not CommonJS	20,977
Others	5,013
TOTAL	1,055,131

Table 3: NPM measurement experiment overall status

fully reduced packages for both reduction modes. From all packages that successfully passed the validation test after reduction, more than third has 100% test coverage and almost forth have coverage between 90-99.9% for both coarse and fine-grain reductions, as shown in Table 2. This shows that Mininode can successfully reduce the packages without breaking the intentional behavior.

7.2 Attack Surface Reduction in NPM

Experiment Setup. The setup and stages of the measurement experiment are shown in Figure 3. First, we collected all package names from NPM. Second, we tried to install the production version of all packages and to run reduction logic on successfully installed ones. Finally, we analyzed the results and measured the vulnerabilities and their reduction.

We gathered all package names from NPM using the open-source package *all-the-package-names* [12] that contains the list of all package names sorted by dependent count. The list contained 1,055,131 package names from NPM as of 19th September 2019.

After gathering all the package names, we tried to install and reduce packages using the coarse-grain reduction method. Table 3 shows that only 672,242 out of 1,055,131 were successfully installed and reduced. Table 3 lists the most common reasons why not all of the packages were analyzed. Top two most common reasons are: (1) installed packages are not Node.js application, which means they are not intended to run on the server-side, e.g. theme's CSS files; (2) packages that can not be reduced with Mininode, due to non-resolvable dynamic import. One interesting failed category is packages' for which entry point is not CommonJS, e.g. ES6, or even not JavaScript file, e.g. TypeScript, JSON and so on.

In the fourth step, as shown in Figure 3, we gathered a

	Number
Removed fs built-in module	549,254
Removed net built-in module	623,646
Removed http built-in module	606,981
Removed https built-in module	614,030
Percentage of removed JavaScript files	79.1%
Percentage of removed LLOC	90.5%
Percentage of removed exports	90.4%
TOTAL	672,242

Table 4: NPM measurement experiment results

vulnerability database from *snyk.io* [16] and mapped vulnerabilities with packages by calculating if specific vulnerable dependency is part of the dependency chain inside the package. In addition to mapping vulnerability, we calculate if Mininode removed the particular vulnerability during the reduction process. We consider that a specific vulnerable dependency is removed if Mininode removes all source files from it. Otherwise, we say that the package still depends on vulnerable dependency. Note that this is a conservative approach and gives us a *lower bound* reduction number because certainly Mininode *may* have removed a vulnerable file from vulnerable dependency, and left only safe files.

Results. The NPM measurement experiment reduction results are shown in Table 3. As discussed earlier, only 672,242 out of 1,055,131 were successfully installed and reduced. From all successfully installed and reduced packages, Mininode restricted access to `fs` built-in module in 81.7% packages, and it also restricted access to network-related built-in modules such as `net`, `http`, `https` in 92.8%, 90.3%, 91.3% packages, respectively. We discussed how Mininode restricts access to built-in modules in Section 6.1.

One question we tried to answer during the NPM measurement experiment was how significant is the severity of bloated code in NPM packages. To answer this question, we calculated the relationship between declared and installed dependencies of the packages. On average, successfully analyzed packages declared 1.9 dependencies but installed 27.3 dependencies, which means NPM installed x14 times more dependencies than declared. This behavior is the result of the transitive dependency installation process discussed in Section 2.2. On NPM public registry, the package's detailed information shows the number of declared, i.e. direct dependencies, but not the number of actually installed dependencies. As a consequence, developers may choose packages with lower declared, but higher installed dependencies instead of packages with higher declared, but lower installed dependencies.

To give a more detailed insight of the bloatedness of NPM packages, we calculated the ratio between third-party and original code base's logical lines of code. On average, from all code-base, only 6.8% was original code, while 93.2% was external code from third-party dependencies, and from all

LLOC, only 9.5% were left after coarse-grain reduction by Mininode. This result clearly shows that more and more applications are developed as a mash-up of third-party packages and the need for reduction techniques.

We also measured the effectiveness of Mininode in reducing unused vulnerable dependencies from packages. These vulnerabilities are present in the application's code, but are not reachable. In order to get exploited, the attacker needs to chain vulnerabilities together (§ 3), something that might not be always possible. The results of vulnerability reduction analysis in NPM packages are given in Table 5. We used the vulnerability database from `snyk.io`, which contains 1,660 vulnerable packages grouped by categories. In total, we found that 119,433 of packages have at least one active vulnerable dependency by the time of writing. This corresponds to 17.8% of all successfully analyzed and reduced 672,242 packages. Table 5 shows the top ten most common vulnerability categories sorted by the number of unique packages that have a dependency from a specific vulnerability category. For example, 91,184 packages have *at least one* dependency vulnerable to Prototype Pollution. Partially removed column of the Table 5 shows the number of unique packages from which Mininode removed at least one vulnerability of a specific category. For example, if the package `@chrismlee/reactcards` had two vulnerable dependencies from the Arbitrary Code Injection category and Mininode was able to remove one of the vulnerable dependencies, then we count the package as partially removed. On the other hand, the fully removed column shows the number of unique packages where *all* vulnerabilities of the specific category were removed. Also, in Table 5, one can see the percentage of partially and fully removed packages from the total number of vulnerable packages. On average, Mininode was able to partially remove vulnerabilities from across all categories in 13.8% cases, and remove all vulnerabilities in 13.65% cases. In conclusion, Mininode was able to remove at least one vulnerability from 10,618 and remove all vulnerabilities from 2861 unique packages from all 119,433 vulnerable packages.

8 Related Work

Attack Surface Reduction. Howard *et al.* [30] introduced the notion of the attack surface, a way to measure the security of the system. Manadhata [35] generalized Howard's approach and introduced a step by step mechanism to calculate the attack surface of the system. Theisen *et al.* [48] came up with an attack surface approximation technique based on stack traces. There are several attempts both to reduce and to measure the attack surface of the different systems, such as OSes, websites, mobile applications [28, 40, 41, 51]. While all of the above works are related to attack surface reduction, we concentrate on the attack surface reduction of the Node.js applications. Azad *et al.* [21] showed that debloating the web application improves its security. They debloated the PHP

application by recording the web application's code coverage from client-side interaction, which may break the website if rarely used functionality was not triggered during recording step. On the other hand, we use static analysis to create the dependency graph of the application, which covers all use-cases accessible from the application's entry point.

Node.js and NPM Security. Previous researchers on the security of Node.js concentrate more on injection attacks [17, 37, 42] and event poisoning attacks [23–25]. Ojamaa *et al.* was the first to assess the security of Node.js [38]. They conclude that denial of service is the main threat for Node.js. On the other hand, we concentrate on reducing the overall attack surface of Node.js rather than on specific attack or vulnerability.

NodeSentry [26] is a permission-based security architecture that integrates third-party Node.js modules with least-privilege. While NodeSentry also reduces the attack surface by using least-privilege modes for Node.js modules, we approached the problem from a different angle. Mininode removes unused functionality from third-party dependencies instead of restricting their functionality as NodeSentry does.

On the NPM side, researchers try to answer why developers use trivial packages [19] and the security implications of depending on NPM packages [52]. Zimmermann and *et al.*'s results supplement our results that depending on too many third-party packages significantly increase the attack surface [52].

JavaScript Application Analysis. In the past, researchers tried to come up with static [33, 34, 37] and dynamic [6, 36, 37] techniques that help developers with analysis of the application written in JavaScript. Madsen *et al.* [33] focuses on static analysis of JavaScript applications using traditional pointer analysis and use analysis. The key insight of the paper is the idea of observing the uses of library functionality within the application code to better understand the structure of the library code. Madsen *et al.* [34] introduced an event-based call graph representation of Node.js application that is useful to detect various event-related bugs. The advantage of the event-based call graph is that it contains information about listener registration and event emission that can be used to detect dead events and emits. Sun *et al.* [6] introduced a dynamic analysis framework called NodeProf that can be used for profiling, for locating bad coding practices, and for detecting data-race in Node.js applications. Mezzetti *et al.* [36] introduced a technique called type regression testing, which automatically determines if NPM package's update affects the types of its public interface, which eventually will introduce breaking changes for clients. While there exists many other JavaScript static analysis tools, Mininode differs because it mostly concentrates on building dependency graphs to reduce the attack surface.

JavaScript bundlers. Traditionally bundlers are used on the client-side to combine all the source code files into a single file to reduce network requests back to the server. One

Category names	Vulnerable packages	Partially removed	%	Fully removed	%
Prototype Pollution	91,184	5,333	5.85%	3,633	3.98%
Regex Denial of Service	42,163	3,930	9.32%	1,228	2.91%
Denial of Service	21,312	403	1.89%	370	1.74%
Uninitialized Memory Exposure	6,433	690	10.73%	592	9.20%
Arbitrary Code Execution	5,324	413	7.76%	396	7.44%
Cross-Site Scripting	5,142	665	12.93%	590	11.47%
Arbitrary Code Injection	3,451	1,715	49.70%	1649	47.78%
Remote Memory Exposure	3,323	16	0.48%	15	0.45%
Arbitrary File Overwrite	3,240	383	11.82%	381	11.76%
Information Exposure	3,088	47	1.52%	47	1.52%

Table 5: Common vulnerability categories and their reduction results. Some vulnerabilities might not be exploitable since their code is not directly reachable and it might not be possible to chain the vulnerabilities due to additional constrains.

of the most popular bundlers is *webpack* [18], that supports plugins and different file types, *e.g.* CSS, HTML. While the latest version of *webpack* can perform dead-code elimination, which is eliminating declared but unused functions and variables, Mininode removes exported functionalities that are never used outside the module, in addition to dead-code elimination. Another popular bundler is *rollup* [15] which can also remove unused exported functions from modules. However, *rollup* works only for ES6 module system, while Mininode was designed to work with CommonJS module system which is the most widely used in NPM. There are open-source plugins for both *webpack* and *rollup* tools that try to convert CommonJS module into ES6 module, but to our best of knowledge, they do not try to resolve the dynamic challenges that Mininode resolves (see §5 and §6). We envision that our work will be integrated into existing JavaScript bundlers.

9 Limitations

In this section, we discuss some of our evaluation and implementation limitations. First, using a test coverage metric to detect if Mininode breaks the original behavior can be misleading. For example, in the case of dynamic code generation, *i.e.* `eval`, test coverage may give 100% coverage even if it is not covering all functions. However, we argue that test coverage is the most appropriate mechanism that we can use to automatically perform a large-scale evaluation.

Second, we employed the *snyk.io* database in our vulnerability analysis measurement instead of the well-established CVE-DB or NIST. Unfortunately, despite the high quality of reports, both contain less number of reports related to third-party Node.js package vulnerabilities [32].

Third, the dynamic nature of JavaScript is a well-known challenge for static analysis. In this paper we tried to solve some Node.js specific challenges, such as *dynamic import*, and defining *aliases*, by using static analysis. However, there are challenges that cannot be easily resolved with static analysis. For example, one of those challenges is dynamic code gen-

eration using various JavaScript APIs, *e.g.* `eval`, `Function`, `setTimeout`. Another challenge is patching Node.js specific APIs, *e.g.* `require`, as shown in Listing 8. In this case, Mininode will not be able to resolve a module inside a different folder, because it uses an unpatched version of `require`.

```

1 // patching the require
2 require = function(arg) {
3   return {mocked: true};
4 }

```

Listing 8: Example of patching the `require()`

A solution to this challenge can be to dynamically execute the patched code in Mininode to resolve the dynamically required module. Another approach is to forbid patching of `require` function in Node.js application by creating a constant global object `require` that can be accessed by all modules. This way, the function wrapper (See Listing 1) discussed in Section 2.1 does not need to pass `require` as an argument.

10 Conclusion

In this paper, we presented a detailed evaluation of excessive functionality in Node.js applications. We presented a tool, called Mininode, that measures and effectively removes unnecessary code and dependencies by statically analyzing Node.js applications. We conducted an extensive analysis of 672,242 packages listed in the NPM repository and found 119,433 of them to have at least one vulnerable module dependency. Our tool is capable of statically removing all vulnerable dependencies from 2861, and removing partially from 10,618 applications. In addition to removing vulnerabilities, Mininode was able to restrict access to the file system for 549,254 packages. We envision our tool to be integrated into the building process of Node.js applications. Mininode is publicly available at <https://kpravelos.com/projects/mininode>.

Acknowledgments

We would like to thank our shepherd, Johannes Kinder, and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the National Science Foundation (NSF) under grant CNS-1703375.

References

- [1] Babel JavaScript compiler. <https://babeljs.io/>.
- [2] CVE-2020-7681, marscode vulnerability. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7681>.
- [3] CVE-2020-7682, marked-tree vulnerability. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7682>.
- [4] CVE-2020-7687, fast-http vulnerability. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7687>.
- [5] Debug. <https://www.npmjs.com/package/debug>.
- [6] efficient dynamic analysis for node.js.
- [7] Esgodegen. <https://github.com/estools/escodegen>.
- [8] Esprima. <https://esprima.org/>.
- [9] Module Counts. <http://www.modulecounts.com/>.
- [10] Node.js. <https://nodejs.org/en/>.
- [11] Node.js Documentation, ECMAScript Modules. <https://nodejs.org/api/esm.html#esm-ecmascript-modules>.
- [12] NPM package: all-the-package-names. <https://www.npmjs.com/package/all-the-package-names>.
- [13] Npm package: detective. <https://www.npmjs.com/package/detective>.
- [14] Official Babel Documentation. <https://babeljs.io/docs/en/config-files#project-wide-configuration>.
- [15] RollupJS. <https://rollupjs.org/guide/en/>.
- [16] Snyk: Vulnerability DB. <https://snyk.io/>.
- [17] understanding and automatically preventing injection attacks on node.js. Technical report.
- [18] webpack. <https://webpack.js.org/>.
- [19] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? An empirical case study on npm. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017.
- [20] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [21] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the USENIX Security Symposium*, 2019.
- [22] Matthew Baxter-Reynolds. Here's why you should be happy that microsoft is embracing node.js. <https://www.theguardian.com/technology/blog/2011/nov/09/programming-microsoft>.
- [23] James Davis, Christy Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.
- [24] James Davis, Gregor Kildow, and Dongyoon Lee. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *Proceedings of the ACM European Workshop on Systems Security*, 2017.
- [25] James Davis, Eric Williamson, and Dongyoon Lee. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *Proceedings of the USENIX Security Symposium*, 2018.
- [26] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.
- [27] Emily Mitchell. Support for Node.js when you need it. <https://developer.ibm.com/articles/support-offering-for-nodejs/>.
- [28] Sumit Goswami, Nabanita Krishnan, Mukesh Verma, Saurabh Swarnkar, and Pallavi Mahajan. Reducing Attack Surface of a Web Application by Open Web Application Security Project Compliance. *Defence Science Journal*, 2012.

- [29] TJ Holowaychuk. NPM package: requires. <https://www.npmjs.com/package/requires>.
- [30] Michael Howard, Jon Pincus, and Jeannette M. Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*. Springer, 2005.
- [31] Joel Kemp. NPM package: dependency-tree. <https://www.npmjs.com/package/dependency-tree>.
- [32] Sherif Koussa. 13 tools for checking the security risk of open-source dependencies. <https://techbeacon.com/app-dev-testing/13-tools-checking-security-risk-open-source-dependencies>.
- [33] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the ACM Joint Meeting on Foundations of Software Engineering*, 2013.
- [34] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices*, 2015.
- [35] Pratyusa K Manadhata, Kymie M Tan, Roy A Maxion, and Jeannette M Wing. An Approach to Measuring A System’s AttackSurface. Technical report, CMU-CS-07-146, 2007.
- [36] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in Node.js libraries. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2018.
- [37] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of Node.js applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
- [38] Andres Ojamaa and Karl Dūūina. Assessing the security of Node.js platform. In *Proceedings of the IEEE International Conference for Internet Technology and Secured Transactions*, 2012.
- [39] Paypal Engineering. Node.js at PayPal. <https://medium.com/paypal-engineering/node-js-at-paypal-4e2d1d08ce4f>.
- [40] Sebastian Ruland, Géza Kulcsár, Erhan Leblebici, Sven Peldszus, and Malte Lochau. Controlling the Attack Surface of Object-Oriented Refactorings. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2018.
- [41] Mohamed Shehab and Abeer AlJarrah. Reducing Attack Surface on Cordova-based Hybrid Mobile Apps. In *Proceedings of the International Workshop on Mobile Development Lifecycle*, 2014.
- [42] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2018.
- [43] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The Long “Taile” of Typosquatting Domain Names. In *Proceedings of the USENIX Security Symposium*, 2014.
- [44] The npm blog. ‘crossenv’ malware on the npm registry. <https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry>.
- [45] The npm blog. Details about the event-stream incident. <https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry>.
- [46] The npm blog. kik, left-pad, and npm. <https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm>.
- [47] The npm blog. Why we created npm enterprise. <https://blog.npmjs.org/post/183073931165/why-we-created-npm-enterprise>.
- [48] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating Attack Surfaces with Stack Traces. In *Proceedings of the IEEE International Conference on Software Engineering*, 2015.
- [49] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *Proceedings of the IEEE Internet Computing*, 2010.
- [50] Erik Trickel, Oleksii Starov, Alexandros Kapravelos, Nick Nikiforakis, and Adam Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *Proceedings of the USENIX Security Symposium*, 2019.
- [51] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2018.

- [52] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the USENIX Security Symposium*, 2019.

Evaluating Changes to Fake Account Verification Systems

Fedor Kozlov[†], Isabella Yuen[†], Jakub Kowalczyk[†], Daniel Bernhardt[†], David Freeman[†],
Paul Pearce^{†‡}, and Ivan Ivanov[†]
[†]Facebook, Inc
[‡]Georgia Institute of Technology

Abstract

Online social networks (OSNs) such as Facebook, Twitter, and LinkedIn give hundreds of millions of individuals around the world the ability to communicate and build communities. However, the extensive user base of OSNs provides considerable opportunity for malicious actors to abuse the system, with fake accounts generating the vast majority of harmful actions and content. Social networks employ sophisticated detection mechanisms based on machine-learning classifiers and graph analysis to identify and remediate the actions of fake accounts. Disabling or deleting these detected accounts is not tractable when the number of false positives (i.e., real users disabled) is significant in absolute terms. Using challenge-based verification systems such as CAPTCHAs or phone confirmation as a response for detected fake accounts can enable erroneously detected real users to recover their access, while also making it difficult for attackers to abuse the platform.

In order to maintain a verification system’s effectiveness over time, it is important to iterate on the system to improve the real user experience and adapt the platform’s response to adversarial actions. However, at present there is no established method to evaluate how effective each iteration is at stopping fake accounts and letting real users through. This paper proposes a method of assessing the effectiveness of experimental iterations for OSN verification systems, and presents an evaluation of this method against human-labelled ground truth data using production Facebook data. Our method reduces the volume of necessary human labelled data by 70%, decreases the time necessary for classification by 81%, has suitable precision/recall for making decisions in response to experiments, and enables continuous monitoring of the effectiveness of the applied experimental changes.

1 Introduction

Online Social Networks (OSNs) enable people to build communities and communicate effortlessly. With the proliferation of social media usage, OSNs now play a role in the lives

of billions of people every day. The largest social networks—Facebook, Twitter, LinkedIn, and Instagram—provide a broad set of features enabling more than two billion people to share news, media, opinions, and thoughts [12, 49]. The scale and scope of these OSNs in turn attract highly motivated attackers, who seek to abuse these platforms and their users for political and monetary gain [3].

The prevalence, impact, and media coverage of harmful social media accounts has increased commensurately with the growth of the platforms [8, 28]. A key contributor to this problem is *fake accounts*—accounts that do not represent an authentic user, created for the express purpose of abusing the platform or its users.

Recent research estimates as much as 15% of all Twitter accounts to be fake [51], and Facebook estimates as much as 4% of their monthly active users to fall into this category [11]. These fake accounts post spam, compromise user data, generate fraudulent ad revenue, influence opinion, or engage in a multitude of other abuses [14, 15, 38, 44, 48].

The variety of behaviours exhibited by fake accounts—especially those controlled by humans—makes building accurate detection systems a challenge. On a platform with billions of active users, a detection system with even 99% precision would incorrectly identify hundreds of thousands of users *every day* as malicious. It follows that OSNs require remediation techniques that can tolerate false positives without incurring harm, while still providing significant friction for attackers.

A common OSN remediation technique is to enroll fake accounts detected by a detection system into a *verification system* [17, 33] aimed at blocking access to the OSN for fake accounts and providing a way to recover an account for legitimate users. These systems are composed of *challenges* which prompt identified users to provide some additional information such as phone numbers, recent activity, or identity verification. These challenges—of which the best known example is a CAPTCHA [53]—take the form of challenge-response tests that are designed to be easy for real users to pass, but difficult for attackers to solve. Verification systems

have numerous advantages over direct disabling of accounts. They provide a soft response that is tolerant of false positives: a real user classified as potentially fake has semi-automated means of changing the classification result without substantial impact on their engagement. The challenges themselves provide an opportunity to collect additional signals about the user (e.g., time-to-solve), which can aid in further investigation, re-classification, and remediation. The strength (friction) of the challenge can be scaled based on initial classification confidence of the detection system.

Despite these advantages, attackers can adapt to overcome the friction posed by verification system challenges [27, 35, 41]. It follows that continuously iterating on the design of those challenges and being able to measure the *effectiveness* of the iterations over time is an important component of improving fake account defences, which has not yet been addressed in the research literature.

We seek to understand *iteration effectiveness*: the degree to which a new or improved challenge is more successful in both stopping fake accounts and letting real users through. To compare effectiveness, we subject pools of accounts to two different experiences in an A/B experiment and compute the change in the proportion of fake and real accounts that managed to successfully pass the verification process. This computation is particularly challenging as it involves determining the true nature of a set of users that were already identified as fake (with high-probability) by an in-production detection framework. To aid in classification one could leverage human labelling of accounts at various stages within and after the verification process. However, relying on human labelling limits the scale and speed of experiments, especially when we require that: many experiments can be run at the same time; we support *backtests*, a technique where some flows are withheld from a small proportion of the population after they have become the default experience, in order to gauge adversarial response; experiments must be powerful enough to show results on different user segments (e.g., platform, locale).

To enable such classification at scale and across such requirements, our approach is to replace the majority of human labelling with automated techniques having precision/recall suitable for both making decisions on the experiments and continuously monitoring the effectiveness of the applied experimental changes.

Our contribution: In this work we develop an automated, scalable method of assessing the effectiveness of experimental iterations for OSN verification systems. A important insight is that we only need weak labels (i.e., “likely” labels) in order to enable rapid experimentation.

Our approach, which we call the *Post Authentication State Model* (PAS), reproduces in an automated way the process that human investigators use to determine the authenticity of an account. PAS requires accounts to be observed for a certain period of time after the verification process in order to collect

additional signals, after which they are evaluated against a continuously retrained machine-learned ensemble decision tree of account behaviours. Using this model to evaluate test and control groups of accounts that pass the verification system allows us to determine the change in post-verification fake/real distributions and ultimately how successful an introduced change is at improving the system’s effectiveness. Section 3 provides an overview of Facebook’s verification system and relevant background. Section 4 discusses the design of this model and several variants. We assess our approach with experiments conducted on Facebook’s production verification system, described in Section 5. Our system: enables rapid A/B experimentation; supports an arbitrary number of backtests of the experimental changes, allowing us to continuously monitor the effectiveness of the improvements and adversarial response over time; supports a variety of verification system challenges.

We deployed our approach in a real-world setting at Facebook to assess its potential effectiveness. Our approach, PAS, provided useful signal on whether accounts clearing the verification system were real or fake; it vastly out-performed random assignment, achieving precision over 70% and recall over 60% for all three classes. This approach reduced the volume of human labelling for the life cycle of an experiment by 70%, and the labelling frequency from continuous to a single post-experiment operation. Practically, we showed that our approach could reduce the time necessary for classification by up to 81%. This reduction in human effort allowed Facebook to run more experiments in parallel, improving the agility and scale of their experimentation methods.

Furthermore, the deployed model completely automated the backtests of successfully launched experiments. Thanks to automated backtesting, three instances of adversarial adaptation to the experimental changes were discovered, allowing the Facebook team to quickly find appropriate mitigations.

Out-of-scope: In this work, we focus on classification of fake and real accounts that were already detected by an in-production detection framework and were able to pass challenges in OSN verification systems, such as CAPTCHA and phone confirmation. Automated classification of these accounts enables an assessment of experimental iterations for OSN verification systems in order to improve real user experience and increase friction for fake accounts. Based on description above, we consider the following areas out of scope of this work: improvements to efficiency and accuracy of existent fake account detection systems and methods; measurement of recall and precision of fake account detection systems; and improvements made to verification systems.

2 Related Work

There is a large literature examining fake accounts in social networks. This work touches on understanding what the ac-

counts are doing (e.g., scamming, impersonation, etc.), methods for detecting fake accounts, and providing techniques (e.g. CAPTCHA) to effectively address detected fake accounts.

2.1 Types of Fake Accounts

Fake accounts (sometimes called *sybils* [56]) can be divided into three broad classes: *automated*, *manual*, and *hybrid* [7, 21]. Automated fake accounts—*social bots*—are software-controlled profiles that can post and interact with legitimate users via an OSNs’s communication mechanisms, just like real people [38]. Usually, social bots are created at scale via automated registration of new accounts in OSNs. The types of abuse caused by social bots varies. There have been instances of social bots that participate in organised campaigns to influence public opinion, spread false news, or gain personal benefits [2, 44]. Recently, social bots have targeted and infiltrated political discourse, manipulated the stock market, stolen personal data, and spread misinformation [15].

In contrast, manually driven fake accounts (MDFA) are set up for a specific purpose without using automation, and are then operated manually by attackers to gain personal benefit [20], push propaganda [28], or otherwise harm users of the platform. The close similarity between actual users and MDFAs breaks traditional at-scale detection techniques which focus on identifying automated behaviours.

Hybrid fake accounts (sometimes called *cyborgs* [7]) include fake accounts driven by bot-assisted humans or human-assisted bots. In practice, sophisticated attackers may choose a mix of tactics for running cyborg fake accounts. Cyborgs are often used for the same purposes as social bots, such as spam and fake news [39].

2.2 Detecting Fake Accounts

The topic of detection of fake accounts is actively explored in recent literature. Research has mostly focused on the design and measurement of detection systems with the purpose of increasing precision and recall. Detection frameworks can be based on different methodologies.

Graph-based and sybil detection focuses on exploring connections between identities and their properties inside social graph to detect fake accounts [9, 23, 56]. A typical example of graph-based sybil detection framework is Sybilguard [58]. The detection protocol of this framework is based on the graph among users, where an edge between two users indicates a human-established trust relationship. Malicious users can create many identities but few trust relationships. Therefore, there is a disproportionately-small “cut” in the graph between the sybil nodes and the honest nodes. Other examples of the detection frameworks based on this methodology that use various algorithms and assumptions about social graph are Sybillimit [57], Sybilinfer [10], SybilRank [5].

Behaviour-based and spam detection employs rule-based heuristics to detect fake accounts. An example of such heuristic is rate limits on specific user activity such as comments and posts and anomalies of such activities. This methodology focuses on high precision to avoid high false positive rate in detection and usually shows low recall [45, 52, 54, 59]. Another example of behaviour-based detection system is SynchronoTrap. This system employs clustering of accounts according to the similarity of their actions to detect large groups of abusive accounts [6].

Machine learning detection frameworks use machine learning models to detect fake accounts [16, 24, 47, 55]. Machine learning models are usually trained based on human labelled data or high precision proxies and utilize an extracted set of user’s behavioral features. One of the first examples of such machine learning detection frameworks was proposed by Stein et al. [43]. There are two main downsides of this methodology: it is challenging to properly design features that are resistant to adversarial response, and the process of collecting high precision training data based on human labelling is expensive.

Digital footprint detection employs digital footprints of OSN users to detect fake and malicious accounts across different social networks. A digital footprint is generated based on publicly available information about a user, such as user-name, display name, description, location, profile image and IP address [29, 46].

Described methodologies of fake account detection and detection frameworks can’t be directly used to measure effectiveness of the improvements in verification systems for fake accounts because users in verification systems are already classified as fakes by detection frameworks. However, in the proposed approach, we use learnings and techniques from machine-learning, graph-based and behaviour-based detection methodologies.

2.3 Remediating Fake Accounts

Once fake accounts are detected, social networks must decide how to respond. Typical actions that a social network might take on detected fake accounts include disabling or deletion. Such responses might be appropriate in some particular cases, where the approximate cost of abusive actions taken by fake accounts and the cost of disabling a real user can both be established. In such cases, the detection framework owner can use this information to make a trade-off between recall and precision [36]. However, representing user actions and cost in financial terms typically will only apply to very narrow scenarios like e-commerce transactions.

In order to allow incorrectly detected real users to regain access to the system, OSNs employ verification systems and challenges. There are numerous types of challenges, including email verification, CAPTCHA resolution, phone confirmation, time and IP address restrictions, challenge questions

and puzzles, manual review of the account, ID verification, facial/voice recognition, and challenges based on liveness detection systems [1, 25, 33, 42, 50]. Most prior work related to verification systems for fake accounts covers new types of verification challenges [22, 30–32] or ways to bypass these systems [26, 60]. This paper is focused on the effectiveness measurement of the improvements in verification systems for fake accounts, for which there is no prior exploration.

3 Background

In this section we frame the overall space of fake account verification systems, outline the metrics used to evaluate the effectiveness of such systems, and discuss prior systems used at Facebook.

3.1 Verification Systems and Clearance

The purpose of OSN fake account verification systems is to block access to the OSN for accounts detected by fake account detection systems; present those accounts with various challenges that allow them to identify themselves as legitimate; collect additional signals by means of those challenges; and ultimately make a determination if an account is real or fake.

An account that is determined to be real is said to “clear” the challenge. Figure 1 shows the structure of an OSN fake account verification system such as the one used at Facebook. A particular path an account takes through the system, which involves passing one or more challenges, is called a *flow*. A flow is divided into *flow points*, or *steps*, which describe the current state of the account within the verification system. Each step can have a number of outcomes, which result in transitions to different steps in the flow or back to the same step. Thus the verification system is essentially a set of possible flows on a directed (possibly) cyclic graph, where the nodes are the steps and the edges are the possible step transitions.

A step is most often associated with a user interface (UI) screen that either requires user input or contains some information for the user, for example an introduction step that explains the reason for being enrolled into the verification system. Some steps contain only a piece of business logic and are invisible to the user. An example of such a step is the *challenge chooser*, which contains rules to decide whether the user has provided sufficient information to determine the authenticity of their account; if the answer is negative, this step will also decide which challenge to show the user. In the context of the flow graph, a challenge is represented as group of one or more steps that need to be completed to proceed forward through the flow.

Each challenge and the steps within it present variable friction to the user, defined as the degree of difficulty in solving the challenges or proceeding through a given step. This friction causes two observable phenomena in the flow graph.

The first phenomena is *churn*, defined as the number of users which do not proceed further through the flow in a given step, which reveals how restrictive a step is for a user. The second phenomena is *anomalies in step completion*, such as spikes or long term drift, which reveal, for example, that bad actors have become proficient at solving the challenge or that there is a loophole in the system being exploited by attackers.

To measure these phenomena, Facebook uses a “funnel logging” system. This system tracks transition events through the flow graph—when a user proceeds from one step to another step, receives a challenge, starts or finishes the verification system flow. Figure 1 shows such events as dots labelled with dashed boxes. Along with those transition events, event meta-data such as country, device, or user age are logged in order to be able to understand how clearance rates vary across user segments.

Funnel logging allows us to calculate clearance rate metrics that quantify the overall friction for the step, challenge, or verification system as a whole. We can also calculate these metrics for different sub-populations or segments of users. For a specific subpopulation segment Y , enrolled on day d_e , which cleared step s on day d_c , we define the *step clearance rate* C as:

$$C(d_e, d_c, s, Y) = \frac{|d_e, d_c, s, Y|}{|d_e, Y|},$$

where $|\cdot|$ denotes the number of users in a population defined by the given variables. The step clearance rate can be used to calculate the end-to-end challenge or system clearance rate by using the last step of the challenge or flow, respectively, as the input to s .

Using data from the funnel logging system, it is possible to monitor churn for each step and detect anomalies in the clearance rate metrics for specific user segments. The spikes or drops in clearance rate metrics can be an early signal of a bot attack or a bug in the verification system.

However, since our goal is be able to identify fake accounts that pass verification challenges and can be ultimately operated by attackers with a range of skills, clearance rate alone is not sufficient to fully capture the effectiveness of a set of challenges or the verification as a whole. We need further techniques which have the power to distinguish between real and manually driven fake accounts clearing verification system flows.

3.2 Label and Metric Definitions

We examine the performance of our classification models for distinguishing between fake and real accounts by comparing our classifications to *expert manually labelled accounts*. In order to establish if an account is fake, Facebook uses a team of specialists to review accounts. The reviewers look for specific signals that can indicate whether a account is real or fake, and using these signals ultimately label each account. For the

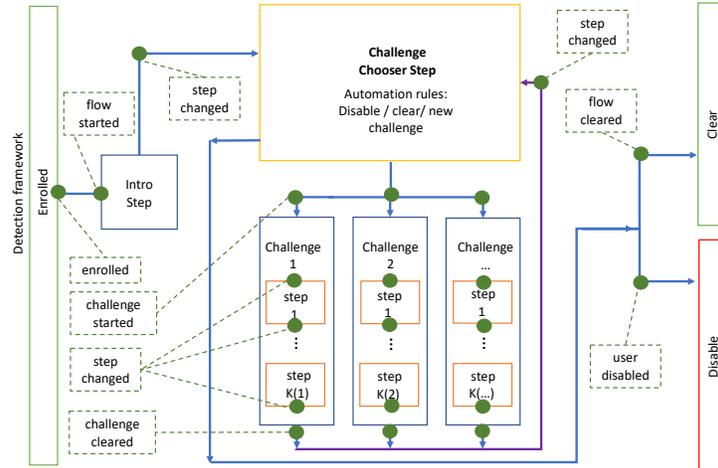


Figure 1: Flow graph showing Facebook’s fake account verification system. Logging events (“funnel logging”) are indicated as dots labelled with the name of the event in the dashed box.

purposes of this work we treat such labelling as *ground truth*. We define three account labels:

- **Abusive:** The account has intent to abuse the platform, including human-driven abuse.
- **Benign:** The account is authentic/real.
- **Empty:** There is *not yet* enough information to classify.

The definition of what constitutes abusive and benign behaviour is specific to the OSN. For example, at Facebook, these labels are defined by the Community Standards document [13].

Human labels are robust and reliable, but not perfect. For example, it is possible that an account’s label might change over time, e.g. empty accounts could be created en masse and then sold days/weeks/months later to individuals who operate the account manually for abusive purposes.

The terms *fake* and *abusive* both refer to fake accounts. The terms *benign*, *authentic*, and *real* all refer to real users. The *prevalence* of a class $Pv(t_i, Y)$ is defined as the true proportion of accounts of class t_i in the overall population Y . Prevalence is typically measured through human labelling on a random sample of population Y , taking care to account for bias in the dataset (e.g., orders of magnitude more good than bad).

The ultimate goal of this work is to enable more rapid and computationally cost effective experiment iteration, and our strategy is to develop systems that can approximate expert human labelling. Section 4 describes several candidate models for classifying users clearing verification flows. The outputs of our models are called *proxy labels*. We evaluate our models based on the *precision* and *recall* [40] of these labels; specifically, for model m which classifies users into classes t_1, t_2, \dots, t_n , we denote the precision and recall of m for class t_i over population Y by $P(m, t_i, Y)$, $R(m, t_i, Y)$, respectively.

We also use the F_1 score [37] of m for class t_i , over population Y , denoted $F_1(m, t_i, Y)$. This score is defined as the harmonic mean of precision and recall:

$$F_1(m, t_i, Y) = \frac{2 \cdot P(m, t_i, Y) \cdot R(m, t_i, Y)}{P(m, t_i, Y) + R(m, t_i, Y)}$$

Both precision and recall are important for classifying users clearing verification systems. High recall across all classes is important as there is limited utility in precisely identifying authentic users if the identified set is only a small fraction of the population. This consideration is equally important for the abusive population, as we will demonstrate in Section 3.3. On the other hand, low precision is unacceptable as it could lead us to believe we are helping authentic users to clear when actually we are helping both authentic and fake users.

The F_1 score gives an overall quality indicator in cases where there is an unequal distribution of fake/real classes, and/or the relative costs of false positives and false negatives are different; both of these conditions hold in fake account verification problems.

A key insight in our examination of this space is that any model that performs better than random assignment will provide useful insight. However, higher precision and recall means we can be more confident in the model thus reducing classification time and human labelling volume. For example, a model with near perfect precision and recall could replace human labelling altogether, whereas a model that is only slightly better than chance could be used in data analysis to support hypotheses but could not be used to accurately measure the effects of changes to real or fake users.

The methods described in this work also use some time delay to accrue signal. We use *time to classification* to refer to the time delay between a user clearing the verification system and enough signals being collected for a label to be assigned.

3.3 Prior Art: BOT Classification Model

The goal of this work is to enable rapid iteration of verification challenge systems, and to that end, we require metrics to

Classification	Label	Precision	Recall	F_1 score
Bot	abusive	86%	6%	12%
Non bot	benign/empty	59%	99%	74%

Table 1: BOT model classification results for the verification system flow.

quickly assess account clearance rates with limited human labelling overhead.

Prior to this work Facebook employed a high precision bot identification model to generate proxy labels and divide users clearing the challenge into “bot” and “non-bot” classes (in addition to numerous other detection and classification systems). This model, which we denote BOT, uses as features metadata collected from fake account detection. In particular, it is often possible to detect a subset of abusive accounts through very high precision rules. When such a rule is triggered the BOT model predicts a fake account, and in all other cases it predicts a non-bot account. Because of this definition, the non-bot class can include a significant proportion of bots that were not detected by the high-precision rules. Applying this model to the clearance rate definition yields the bot proxy clearance $C_b(d_e, d_c, s, Y)$.

Given our goals and requirements, the BOT clearance rate C_b is a potentially attractive option for our proxy metric. In order to verify this hypothesis we sampled tens of thousands of accounts that successfully passed the verification system flow in August 2018 and used human labelling to find the volume of abusive, empty and benign accounts for the resulting class. Table 1 shows the label distribution over the BOT model. While $P(\text{bot}, \text{abusive})$ is fairly high, the model would be of limited value because $R(\text{bot}, \text{abusive}) = 6\%$. The majority of users that cleared the verification system flow are ambiguous, as shown by the precision of the non-bot class, $P(\text{bot}, \text{benign} \cup \text{empty}) = 59\%$. Section 5 evaluates BOT further.

The clear downside of C_b is that the non-bot class has low recall for abusive accounts. The “non-bot clearance” label is thus not accurate enough to measure verification system improvements targeted at real users. The rest of this work explores methods that better approximate human labelling ground truth, quickly, and with limited human input.

4 Post Authentication State Model

When running a large number of A/B experiments it quickly becomes prohibitively resource intensive to use human labelling to classify enough accounts clearing various challenges in each variant to get statistically significant results. Requiring expert human labellers also slows down iteration as such labelling jobs take time. A/B experiments are also often segmented by populations of interest (e.g., platform used, country, locale), which again increases the volume of necessary human labelling and reduces iteration frequency. To understand subtle changes in account clearing performance

and metrics, thousands of labels are required per experiment, and possibly also for each population of interest.

In this section we present the Post Authentication State (PAS) model, a method for generating weak (i.e., likely) labels which enable rapid A/B experimentation. PAS can be scaled and is able to classify users more accurately than prior low computational cost high volume solutions (e.g., BOT classification), while allowing both faster classification and far fewer human labels than full-scale human labelling would require. PAS classifies benign users as well as abusive users, and has significantly higher recall of abusive accounts than other methods.

4.1 Overview

OSNs enroll accounts suspected of being fake into a verification system in order to gain further information about their state. The verification system needs to evolve to match the adversarial response of attackers, so OSNs need to run A/B experiments. PAS classifies accounts clearing the verification system, after a time delay, so that we can understand how the A/B experiment affected the clearance rate of each population (Figure 2). Based on results of A/B experiment OSN can evolve its response to the adversarial adaption of detected fake accounts.

PAS is a decision tree model which aims to emulate human labelling decisions, ultimately assigning an account a proxy label [4]. We denote such labels as “states.” PAS is trained and validated against sets of human labelled accounts using out of the box classifiers based on the CART recursive partitioning algorithms such as SciKit Learn *DecisionTreeClassifier* [18, 34]. The model assigns one of three possible states to the classified account: *Good Post Authentication State (GPAS)* for likely real accounts with authentic signals; *Bad Post Authentication State (BPAS)* for likely fake accounts with intent to abuse the platform; *Empty Post Authentication State (EPAS)* for accounts with too little signal post-clearing to yet make a determination.

PAS predicts the labelling outcome based on signals we can automate, for example number of friends. Gupta et al. [19] showed that decision tree models, based on user level signals and behavioural signals, can be effective in classifying real and fake images in OSNs; we extend this approach to possible fake accounts clearing verification challenges in OSNs. We note that *the PAS model is not designed to be a precise classifier*; instead it buckets users clearing into “probably good” and “probably bad” which gives direction to A/B experiments with higher precision/recall.

Adversarial Adaptation: A common problem in the space of abuse detection systems is adversarial adaptation—can attackers learn what signals are used for detection, and evade them? This is not a direct concern for PAS, since this method is not used to take direct actions on accounts clearing verification system flows; rather it is used to aid in A/B experimentation

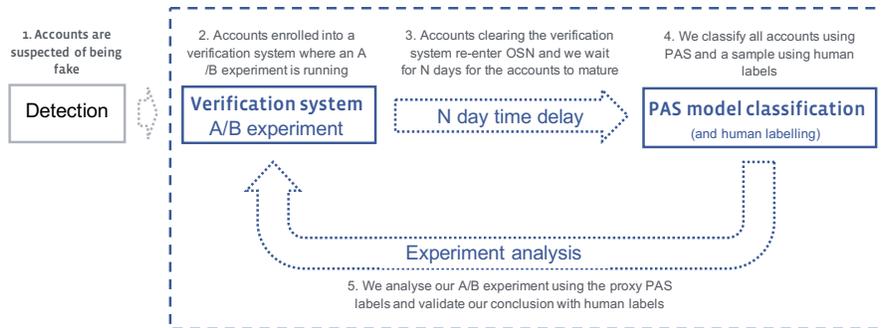


Figure 2: The PAS model as a component of the process to iterate on fake account verification systems.

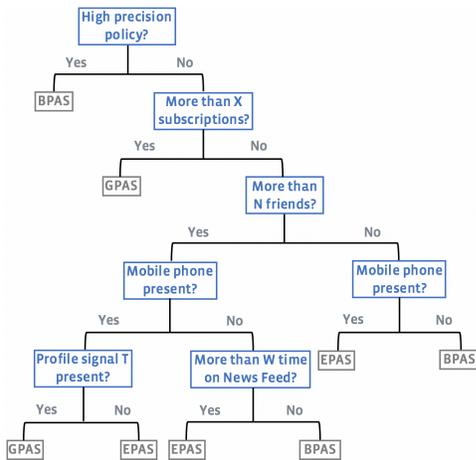


Figure 3: PAS V1 decision tree generated with recursive partitioning (CART). Accounts sent through this flow are ultimately classified with weak labels for A/B experimentation. Threshold values X , N , W , and T are operationally dependent.

and thus product evolution. This means there is no direct mechanism for adversaries to discover which signals to manipulate.

To generate proxy labels we created multiple PAS models iteratively. We started with a simple proof of concept, which showed that we could create a classifier that was better than random assignment but it had flaws in the features selection (Section 4.2). Our next iteration, still a simple proof of concept, used more robust features and was used to understand how the time to classification, or latency, could be improved (Section 4.3). Finally, we created a more accurate model, implemented it in Facebook’s production verification system and showed that it could maintain good performance and allow rapid iteration of the verification system over a 6 month period (Section 4.4).

4.2 PAS V1 and PAS V2: Simple decision trees

The inputs to the PAS model are attributes and behaviours we can associate with the account. Account-level attributes include features such as the number of friends or email domain the account signed up with. Behaviours include post-

clearance activity such as the number of friend requests sent or number of times other users reported the account. For each potential input, we first observed how prominent it was in each labelling population, to understand its potential impact in the construction of a decision tree.

Figure 3 depicts the first PAS decision tree we developed to classify users clearing fake account verification challenges at Facebook. This was a simple decision tree that remained static rather than being retrained. We wanted to understand how this tree performed initially and how it degraded over time. Behavioural signals such as “More than W time on News Feed” correlate to how engaged and how manual the account is, which in turn increases the likelihood that the account is a real user. We leave a specified time period post-clearing to allow these behavioural signals to accrue; it aligns with the period we use to allow labelling signals to accrue before human labelling, and so there is no decrease in the time to classification (Section 5). The specific features in this construction can vary based on OSN use case. For example, “News Feed” could be swapped for another product users engage with in other OSNs. Profile information such as “mobile phone present” could be replaced with other engagement signals such as employment status or current city.

During our evaluation of the first simple PAS model we saw a clear decline in performance of the PAS V1 model over time. This resulted from an important signal (the “high precision policy” in Figure 3) having lost its discriminating power due to changes in the prevalence of the signal in the fake population. We also identified that decision points which are also prerequisites for challenges (e.g., the “having a mobile phone number” signal is a prerequisite for the SMS challenge) create bias in A/B experimentation; since experiments that change the distribution of challenges offered would a priori skew the resulting proxy labels. As a result of these observations we developed a subsequent PAS model, PAS V2, which addressed these limitations.

PAS V2 is structured similarly to PAS V1, constructed again using CART. In this iteration, the “high precision policy” signal is replaced with signals we identified experimentally to be longitudinally stable and have high distinguishing power (Figure 4). Two new signals were added to the tree: one based on how many times the user logged in (behavioural) and one

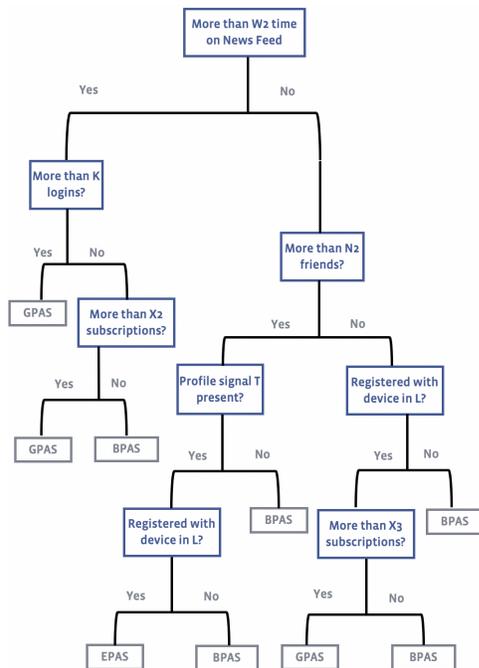


Figure 4: PAS V2 decision tree generated with recursive partitioning (CART). Accounts sent through this flow are ultimately classified with weak labels for A/B experimentation. Threshold values X_2 , X_3 , N_2 , W_2 , K , L , and T are operationally dependent.

based on the device they registered with (account attribute). The delay period post-clearing, used to allow signals to accrue and to calibrate thresholds used for signals such as “more than N friends,” remained the same between the two models.

Section 5 contains a detailed evaluation of PAS V1 and PAS V2 performance.

4.3 Quick PAS: Decreasing time to classification

The simple PAS decision trees use fewer signals than the human labelling trees, and the signals are not contextual. Given this, we hypothesised the time to classification (delay post-clearing) is less critical to PAS than human labelling i.e., decreasing it would not significantly impact precision and recall.

There are two natural ways to decrease the time to classification. The first is artificially limiting the time to classification, running the same model sooner. We assessed the precision and recall of these models when run at truncated delays post-clearing; between 40% and 80% of the full time to classification before human labelling. As hypothesised, reducing the time to classification did not yield significant reductions in precision and recall, even at the shortest time to classification tested.

The second method explored to limit the time to classification was to train a new decision tree with a shorter delay post-clearing and a feature set pruned of time sensitive signals.

We created “Quick PAS,” a reduced-time version of PAS V2 that provides signals more than 5 times faster than PAS V2. Quick PAS has lower thresholds for behavioural signals, such as time on News Feed, and omits some of the signals that take more time to collect, such as having subscriptions. It is important to note that the trade-off in using Quick PAS is not just precision/recall; we are also biasing towards accounts that return to the platform faster than others.

Section 5 evaluates Quick PAS in the context of other PAS models. It also shows the performance of PAS V2 when time to classification is reduced by just over 50%, “Truncated PAS V2.”

4.4 PAS Production: Ensemble decision tree with retraining

The simple PAS decision tree models showed promise in terms of accuracy and latency (time to classification). However, fake account detection and response is an adversarial space; attackers adapt their approach over time to try to evade detection and deceive response verification systems. The consequence is that a simple decision tree model, trained at a particular point in time, will degrade in accuracy as fake accounts evolve. Moreover, training just once makes the model vulnerable to anomalies in the training data.

The next iteration, PAS Production, was developed to address these limitations. PAS Production uses an ensemble decision tree model, to avoid overfitting; it is also retrained every day using a rolling window of training labels from the last few weeks, to retain freshness. This model uses SciKit Learn *BaggingClassifier* combined with *DecisionTreeClassifier*. Like PAS V1 and PAS V2, this model was trained with time to classification the same as the post-clearance delay to human labelling. The goal of PAS Production was to make a more accurate and reliable model, rather than a faster one. A “Quick PAS” could be developed in the same way as described in Section 4.3, by trimming the feature set and training the model with a shorter delay post-clearing.

Additionally, we explored using SciKit Learn probability outputs to gauge uncertainty of the predicted label. Averaging these probabilities for each class in each experiment group can give more signal than taking the most likely class. For example, test groups A and B might have the same number of GPAS (real account) predictions, but group A ’s GPAS accounts might all have higher probabilities associated with them than group B ’s. Averaging the probabilities would reveal this where summing class labels wouldn’t. It’s important to note that probability of a predicted label class can be only be interpreted as confidence of that prediction if the model is well calibrated. SciKit Learn offers calibration functions, such as *CalibratedClassifierCV*, to achieve this.

Section 5 evaluates PAS Production in the context of our other PAS models; for this purpose we restrict our analysis to class predictions and ignore the associated probabilities.

5 Evaluation

In order to assess the effectiveness of the PAS iterations, we evaluated their classification performance against human labelling data on hundreds of thousands of accounts over a period from March 2018 to May 2019. In addition to results, we also identify insights that led to further improvements throughout the evaluation.

The goal of these models is to produce *weak* labelling for use in A/B experimentation, not to produce classification for operational in-production abuse detection. Given this goal we can tolerate medium levels of precision, recall, and F_1 , provided the models perform significantly better than random assignment.

Table 2 presents the results of experiments carried out for each version of the model. The table is divided into three groupings: Baseline results 1-3 (random assignment, BOT, human labelling), iterative developments 4-9 (PAS V1, PAS V2, Truncated PAS V2, Quick PAS), and current deployment 10-11 (PAS Production). The last grouping represents the final iteration of the system and shows significant decreases human labelling volume and improvements over previous models.

5.1 Baselines: Random Assignment, BOT, and Human Labelling

Since we take human labelling to be our ground truth, human labelling provides the benchmark and optimal result for models intending to classify users clearing our verification system (Table 2, Row 3). If we classified users with random assignment, then recall would be 1/3 for each class and precision would be the prevalence of that class in the population of accounts sent for verification (Table 2, Row 1). Random assignment provides a lower bound to compare models against; any model with lower precision and/or recall than random assignment would be detrimental in evaluating experiments.

The BOT model provides a second comparison point. This model uses a high precision signal available from detection to classify users as fake. The signal used is a binary signal which predicts an account to be fake (or BPAS), if it exists for the account. It cannot predict whether an account is authentic (GPAS) or empty (EPAS). Table 2, Row 2 provides the precision, recall, and F_1 scores for BOT. As a result of the signal existing prior to the account clearing fake account verification systems, there is no time delay needed to use it for prediction. We observe that the BOT model's BPAS precision is high, at 86%, but its recall and thus F_1 are low at 6% and 12% respectively. Given the low recall for BPAS and its inability to distinguish the other two classes, we cannot use this model for weak labelling. We require a model that predicts both fake and authentic users because our experiments are designed to prevent fake users from clearing verification systems and help authentic users to do so. Moreover, low recall for fake users means that this model is not suitable for even the subset of

experiments that try to prevent fake accounts from clearing, because it is able to classify too few of them.

5.2 PAS V1

Table 2, Row 4 shows the performance of PAS V1 in March 2018, during its first iteration. The PAS V1 decision tree performed better than random assignment in terms of both precision and recall and was an initial improvement in classification. EPAS (“empty”), the proxy label for accounts with too little signal to mark as authentic or fake, had the poorest precision and recall but represents the population of accounts we are less motivated by in this use case—our primary objectives are to help increase authentic user clearance (GPAS) and decrease clearance of abusive users (BPAS). PAS V1 has a much better precision-recall trade-off for abusive accounts than the bot/non-bot classification. We did not measure the decrease in human labelling as the limitations of PAS V1 necessitated PAS V2.

Table 2, Row 5 shows the performance for PAS V1 in June 2018, three months after implementation. The precision of benign classifications decreased significantly, from 76% to 25%, and recall across both abusive and empty classifications also similarly decreased. F_1 scores dropped for all classes. As discussed in Section 4.2, the “high precision policy” signal had lost its discriminating power due to changes unrelated to our work. These changes motivated the design of PAS V2.

5.3 PAS V2

Table 2, Row 6 shows PAS V2 performance in July 2018, when it was first evaluated. Compared to the degraded scores of PAS V1 from June 2018, PAS V2 shows large improvement in F_1 scores for all classes. In comparison to the initial PAS V1, F_1 score increased for BPAS class and decreased for GPAS classes. Additionally, we've observed that more of its signals have stable distribution over time.

To explore the stability of the system, we reran the evaluation of PAS V2 in September 2018, several months after it was first implemented (Table 2, Row 7). Unlike PAS V1, we did not notice a substantial reduction in performance over time. The main change was that the F_1 score for abusive accounts dropped from 72% to 53%, primarily from abusive precision dropping from 66% to 42%. The drop is caused by changes in the abusive clearance population; fewer accounts were being labelled as abusive, and more were labelled as empty—potentially due to attackers choosing to let accounts “sleep” in response to concurrent, independent work on improved detection.

PAS V2 does not have the same issues as PAS V1 with respect to signals that can be skewed by the verification system itself and none of the underlying signals changed in definition. However the reduction in abusive precision highlights the fact it is necessary to monitor and retrain the PAS decision

Row	Method	Time Period	BPAS Abusive			GPAS Benign			EPAS Empty			Decrease Class. Time	Decrease Human Label Vol.
			Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1		
1	Rand. Assign.	Sep 2018	33%	33%	33%	25%	33%	28%	42%	33%	36%	–	–
2	BOT	Aug 2018	86%	6%	12%	–	–	–	–	–	–	–	–
3	Human Label.	All	100%	100%	100%	100%	100%	100%	100%	100%	100%	0%	0%
4	PAS V1	Mar 2018	65%	61%	63%	76%	78%	77%	47%	51%	49%	0%	–
5	PAS V1	Jun 2018	74%	32%	45%	25%	82%	38%	40%	32%	36%	0%	–
6	PAS V2	Jul 2018	66%	80%	72%	53%	64%	58%	76%	35%	48%	0%	70%
7	PAS V2	Sep 2018	42%	70%	53%	57%	62%	59%	79%	33%	46%	0%	70%
8	PAS V2 Trunc.	Jul 2018	63%	77%	70%	52%	60%	56%	73%	36%	48%	56%	–
9	Quick PAS	Jul 2018	61%	76%	68%	59%	36%	45%	55%	45%	50%	81%	70%
10	PAS Production	Nov 2018	73%	61%	66%	71%	71%	71%	78%	86%	82%	0%	70%
11	PAS Production	May 2019	68%	62%	65%	61%	61%	61%	74%	81%	78%	0%	70%

Table 2: Comparison of PAS models broken down by classification method and validated against human labelling. The first grouping of rows shows idealised and prior methods. The second grouping shows results of intermediate techniques. The third grouping shows results of the final design.

tree model at regular intervals to mitigate risks of changing behaviours in the clearance population.

5.4 Truncated PAS V2 and Quick PAS

To verify our hypothesis about the trade-offs associated with shortened post-clearing delay (Section 4.3), Table 2 (Row 8) measures performance of PAS V2 after truncating the post-clear calculation delay by just over 50%. Compared with PAS V2 evaluated over the same period, the performance of Truncated PAS V2 is only very slightly lower for each class. This experiment confirmed that the post-clearing delay can be reduced without compromising accuracy, which allows us to introduce lower thresholds for behavioural signals and train a decision tree optimised for those changed thresholds and shortened delay. Such changes were codified (beyond a simple reduced threshold) into Quick PAS (Section 4.3).

Table 2, Row 9 shows the performance of Quick PAS in July 2018. Quick PAS has lower F_1 scores in all classes compared to PAS V2. However, benign recall drops and empty recall increases, since the reduced time window limits our ability to collect authentic engagement signals which would ultimately disambiguate an “empty” account from benign for expert human labellers.

5.5 PAS Production

Table 2, Row 10 shows the performance of PAS Production in November 2018. PAS Production strikes the best performance balance between classes: it is the only model to have F_1 scores above 60% for every class. In particular, the Empty (EPAS) F_1 score is much higher than other models, 82% compared with 50% or less from previous models, due to increased recall. This could be a result of the retraining, allowing thresholds to adapt. The Benign (GPAS) F_1 score is also higher than PAS V2’s, 71% compared with 59% or less, due to increased precision. This could be a result of using an ensemble model and not overfitting on the training data. The Abusive (BPAS)

F_1 score is slightly lower than the F_1 score of PAS V2 when it was first developed, 66% compared with 72%. However, this is a much smaller drop than the gain in accuracy for the other two classes and still much higher than random assignment, so we find this acceptable. To verify our hypothesis that PAS Production is more robust than previous PAS models that didn’t retrain, we reran the evaluation of PAS Production six months later in May 2019 (Table 2, Row 11). The precision, recall and F_1 scores of all three classes remained above 60%. The largest drop was for the F_1 score of the Benign (GPAS) class, from 71% to 61%, changing equally in precision and recall. These drops might result from attackers increasing their efforts to appear real over those six months, as far as the automatable signals used in PAS can tell. Our human labelling process relies on more signals, some of which are contextual, and it adapts over time. We are still confident that our human labels represent ground truth.

Our ensemble decision tree, PAS Production, which has been implemented to retrain daily, shows more consistent performance between the three classes and more robustness to time compared with previous models. It has the same time to classification as the labelling process. A lower latency “Quick PAS Production” could be developed to complement PAS Production, to provide earlier signal for A/B experiments.

5.6 PAS Impact

We integrated PAS Production into Facebook’s environment to assess their usefulness in the experimentation process. When a change was introduced into a verification system through an experiment, we used PAS to understand how the change impacted real and abusive accounts clearing the system. In order to understand how experiments impact how accounts flow within a verification system, we used funnel logging event aggregations within challenges to identify the number of accounts attempting and passing challenges, and the time taken. We used the proxy labels assigned by the PAS models, combined with the funnel logging metrics, to support

or refute our hypotheses. If the proxy labels and the additional metrics supported the experiment hypothesis, we would then supplement with additional human labelling to validate results before launching the change.

Decreased Classification Time: Quick PAS showed that we are able to get directional signal on experiments with significant reductions in the time to classification that human labelling requires. This early signal enables us to stop failing experiments earlier or request human labelling validation so that we can launch a change sooner. Quick PAS decreased classification time by 81% whilst keeping accuracy for each class well above random assignment.

Decreased Human Labelling Volume: As outlined in Section 3, the purpose of an OSN fake account verification system is to block access for abusive accounts; and permit benign accounts to re-enter the OSN. Experiments on verification systems will aim to achieve one of these objectives, without harming the other. It is thus necessary to understand how an experiment affects each population and not rely on just the overall clear rates. For example, without further breakdown, an increase in the volume of accounts clearing the verification system cannot be interpreted as achieving the objective of helping benign accounts; as these incremental accounts might be overwhelmingly abusive. A significant amount of labels are required to understand the effects of an experiment at different stages. Accounts have to be labelled early to catch failing experiments sooner. In addition, accounts clearing in subsequent days have to be labelled to mitigate effects of selection bias of the early-stage labelling. Finally, labelling may be required to measure adversarial response several weeks after shipping a feature, using a holdout.

Using the Wald method of binomial distribution, in order to estimate the proportion of accounts in each group that are abusive, benign and empty, to within a 5% error bound, we would need 400 labels per group. Doing this several times per experiment, for multiple experiments per week, would mean tens of thousands of labels are required each week. Human labels are a scarce resource and can't be scaled to support experiments. Pairing the PAS model-produced proxy labels with just one set of validation human labels per experiment, for only those experiments we believe are successful, reduces total human label volume. This method saves early-stage labels on all experiments and it saves all label requirements in clearly negative experiments; as PAS proxy labels give this information. We evaluated labelling volume from July to May 2019. Over this period, Facebook launched and analysed more than 120 experiments. In total, 20,000 human labels were required to be confident about shipping iterations to the fake account verification system. Facebook saved an estimated 50,000 human labels that would have otherwise been required to monitor these experiments. PAS models reduced the volume of human labelling required for experiment analysis by 70% (Table 2). Additionally, as each of the launched experiments required

substantially fewer labels, Facebook could run many more experiments in parallel.

Adversarial Adaptation: In addition to improving efficiency, the models were successfully used for automated monitoring in backtests of launched features. With this framework, Facebook discovered three cases in which the adversaries eventually adapted to the new feature, which would manifest itself as a shift in BPAS prevalence in the population exposed to that feature. This measurement allowed the team working on the verification system to quickly discover the underlying reasons for adaptation and mitigate the problem appropriately.

6 Conclusion and Future Work

We have presented a method for evaluating changes to fake account verification systems, the Post Authentication State (PAS) method. PAS uses a continuously retrained machine-learned ensemble decision tree model that proxies human labelling to classify accounts as abusive and benign faster and with less human labelling than prior approaches. PAS can be used to measure the effectiveness of changes in a verification system over time and to analyse A/B experiments which aim to prevent abusive accounts clearing or help benign accounts to clear the system. At Facebook, PAS reduced the volume of human labelling required for experiment analysis by 70% and decreased the classification time of accounts by 81%. The presented method achieved precision over 70% and recall over 60% for all three classes. PAS has allowed Facebook engineering and data science teams to iterate faster with new features for verification challenges, scale experimentation launch and analysis, and improve the effectiveness of verification systems at remediating fake accounts.

In this paper we have mentioned that *fake account* is a generic term that can cover several types of abusive accounts; a high-level taxonomy would be bots and manually driven fake accounts (MDFA). Being able to further divide our abusive labels and further divide BPAS (our proxy label) into abusive bot and abusive MDFA would greatly help to optimise challenge selection in a verification system. For example, there could be challenges that are trivial for humans and difficult for bots (e.g., a well designed CAPTCHA), and there could be challenges that may be solved by bots but deter humans (e.g., a time-consuming verification). If we were able to measure whether a fake account was a bot or a MDFA then we could assign challenges appropriately.

Finally, we note that our implementation and experiments use the data and infrastructure of a single large online social network, Facebook, and therefore the experimental results might be different for other OSNs. We encourage the research community to apply our approach more broadly to determine to what extent the results and conclusions we have presented in this paper transfer to other areas.

References

- [1] Noura Alomar, Mansour Alsaleh, and Abdulrahman Alarifi. Social authentication applications, attacks, defense strategies and future research directions: a systematic review. *IEEE Communications Surveys & Tutorials*, 99, 2017.
- [2] Alessandro Bessi and Emilio Ferrara. Social bots distort the 2016 us presidential election online discussion. 2016.
- [3] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. The socialbot network: when bots socialize for fame and money. In *ACM CCS*, 2011.
- [4] Leo Breiman. *Classification and regression trees*. Routledge, 2017.
- [5] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 197–210, 2012.
- [6] Qiang Cao, Xiaowei Yang, Jieqi Yu, and Christopher Palow. Uncovering large groups of active malicious accounts in online social networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 477–488, 2014.
- [7] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Who is tweeting on twitter: human, bot, or cyborg? In *ACM CCS*, 2010.
- [8] Nicholas Confessore, Gabriel J.X. Dance, Richard Harris, and Mark Hansen. The follower factory. *The New York Times*, 01 2018.
- [9] Mauro Conti, Radha Poovendran, and Marco Secchiero. Fakebook: Detecting fake profiles in on-line social networks. In *Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2012.
- [10] George Danezis and Prateek Mittal. Sybilinifer: Detecting sybil nodes using social networks. In *NDSS*, pages 1–15. San Diego, CA, 2009.
- [11] Facebook. Community standards enforcement preliminary report, 2018.
- [12] Facebook. Facebook reports second quarter 2018 results, 2018.
- [13] Facebook. Community standards - integrity and authenticity, 2019.
- [14] Nicholas Fandos and Kevin Roose. Facebook identifies an active political influence campaign using fake accounts. *The New York Times*, 07 2018.
- [15] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The rise of social bots. *Communications of the ACM*, 2016.
- [16] M. Fire, G. Katz, and Y Elovici. Strangers intrusion detection - detecting spammers and fake profiles in social networks based on topology anomalies. *ASE Human Journal*, 2012.
- [17] Hongyu Gao, Jun Hu, Tuo Huang, Jingnan Wang, and Yan Chen. Security issues in online social networks. *IEEE Internet Computing*, 2011.
- [18] Raúl Garreta and Guillermo Moncecchi. *Learning scikit-learn: machine learning in python*. Packt Publishing Ltd, 2013.
- [19] Aditi Gupta, Hemank Lamba, Ponnuram Kumaraguru, and Anupam Joshi. Faking sandy: Characterizing and identifying fake images on twitter during hurricane sandy. In *World Wide Web Conference (WWW)*, 2013.
- [20] JingMin Huang, Gianluca Stringhini, and Peng Yong. Quit playing games with my heart: Understanding online dating scams. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.
- [21] Rodrigo Augusto Igawa, Sylvio Barbon Jr, Kátia Cristina Silva Paulo, Guilherme Sakaji Kido, Rodrigo Capobianco Guido, Mario Lemes Proença Júnior, and Ivan Nunes da Silva. Account classification in online social networks with lbca and wavelets. *Information Sciences*, 332:72–83, 2016.
- [22] Anil K Jain, Karthik Nandakumar, and Arun Ross. 50 years of biometric research: Accomplishments, challenges, and opportunities. *Pattern Recognition Letters*, 79:80–105, 2016.
- [23] Jing Jiang, Christo Wilson, Xiao Wang, Wenpeng Sha, Peng Huang, Yafei Dai, and Ben Y. Zhao. Understanding latent interactions in online social networks. *ACM Trans. Web*, 7(4), November 2013.
- [24] Lei Jin, Hassan Takabi, and James B.D. Joshi. Towards active detection of identity clone attacks on online social networks. In *ACM Conference on Data and Application Security and Privacy*, 2011.
- [25] Sam King. Stopping fraudsters by changing products, 2017.

- [26] David Koll, Martin Schwarzmaier, Jun Li, Xiang-Yang Li, and Xiaoming Fu. Thank you for being a friend: an attacker view on online-social-network-based sybil defenses. In *Distributed Computing Systems Workshops (ICDCSW)*, 2017.
- [27] Martin Kopp, Matej Nikl, and Martin Holena. Breaking captchas with convolutional neural networks. In *CEUR Workshop Proceedings*, volume 1885, pages 93–99, 2017.
- [28] Kate Lamb. “i felt disgusted”: inside indonesia’s fake twitter account factories. *The Guardian*, 07 2018.
- [29] Anshu Malhotra, Luam Totti, Wagner Meira Jr, Ponurangam Kumaraguru, and Virgilio Almeida. Studying user footprints in different online social networks. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pages 1065–1070. IEEE Computer Society, 2012.
- [30] Merylin Monaro, Luciano Gamberini, and Giuseppe Sartori. The detection of faked identity using unexpected questions and mouse dynamics. *PLoS one*, 12(5):e0177851, 2017.
- [31] Romklaus Nagamati and Miles Lightwood. Audio challenge for providing human response verification, 2015. US Patent 8,959,648.
- [32] Palash Nandy and Daniel E Walling. Transactional visual challenge image for user verification, 2008. US Patent App. 11/679,527.
- [33] Avani Pathak. An analysis of various tools, methods and systems to generate fake accounts for social media. Technical report, Northeastern University, Boston, Massachusetts, December 2014.
- [34] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.
- [35] Iasonas Polakis, Marco Lancini, Georgios Kontaxis, Federico Maggi, Sotiris Ioannidis, Angelos D Keromytis, and Stefano Zanero. All your face are belong to us: breaking facebook’s social authentication. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 399–408, 2012.
- [36] David Press. Fighting financial fraud with targeted friction, 2018.
- [37] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.
- [38] Chengcheng Shao, Giovanni Luca Ciampaglia, Onur Varol, Alessandro Flammini, and Filippo Menczer. The spread of fake news by social bots. *arXiv preprint arXiv:1707.07592*, 2017.
- [39] Kai Shu, Amy Sliva, Suhang Wang, Jiliang Tang, and Huan Liu. Fake news detection on social media: A data mining perspective. *ACM SIGKDD Explorations Newsletter*, 2017.
- [40] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 2009.
- [41] Saumya Solanki, Gautam Krishnan, Varshini Sampath, and Jason Polakis. In (cyber) space bots can hear you speak: Breaking audio captchas using ots speech recognition. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 69–80. ACM, 2017.
- [42] David J Steeves. Client-side captcha ceremony for user verification, 2012. US Patent 8,145,914.
- [43] Tao Stein, Erdong Chen, and Karan Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, page 8. ACM, 2011.
- [44] Stefan Stieglitz, Florian Brachten, Björn Ross, and Anna-Katharina Jung. Do social bots dream of electric sheep? a categorisation of social media bot accounts. *arXiv preprint arXiv:1710.04044*, 2017.
- [45] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Detecting spammers on social networks. In *Proceedings of the 26th annual computer security applications conference*, pages 1–9. ACM, 2010.
- [46] Gianluca Stringhini, Pierre Mourlanne, Gregoire Jacob, Manuel Egele, Christopher Kruegel, and Giovanni Vigna. Evilcohort: Detecting communities of malicious accounts on online services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 563–578, 2015.
- [47] Enhua Tan, Lei Guo, Songqing Chen, Xiaodong Zhang, and Yihong Zhao. Unik: Unsupervised social network spam detection. In *ACM International Conference on Conference on Information & Knowledge Management*, 2013.
- [48] Kurt Thomas, Chris Grier, Dawn Song, and Vern Paxson. Suspended accounts in retrospect: an analysis of Twitter spam. In *ACM Internet Measurement Conference (IMC)*, 2011.
- [49] Twitter. Investor fact sheet. q2 2018 highlights, 2018.

- [50] Erkam Uzun, Simon Pak Ho Chung, Irfan Essa, and Wenke Lee. rtcaptcha: A real-time captcha based liveness detection system. In *NDSS*, 2018.
- [51] Onur Varol, Emilio Ferrara, Clayton A Davis, Filippo Menczer, and Alessandro Flammini. Online human-bot interactions: Detection, estimation, and characterization. *arXiv preprint arXiv:1703.03107*, 2017.
- [52] Bimal Viswanath, Muhammad Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Towards detecting anomalous user behavior in online social networks. In *USENIX Security*, 2014.
- [53] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311. Springer, 2003.
- [54] Alex Hai Wang. Don’t follow me: Spam detection in twitter. In *2010 international conference on security and cryptography (SECRYPT)*, pages 1–10. IEEE, 2010.
- [55] Cao Xiao, David Mandell Freeman, and Theodore Hwa. Detecting clusters of fake accounts in online social networks. In *ACM Workshop on Artificial Intelligence and Security*, 2015.
- [56] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y. Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Trans. Knowl. Discov. Data*, 8(1):2:1–2:29, February 2014.
- [57] Haifeng Yu, Phillip B Gibbons, Michael Kaminsky, and Feng Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 3–17. IEEE, 2008.
- [58] Haifeng Yu, Michael Kaminsky, Phillip B Gibbons, and Abraham Flaxman. Sybilguard: defending against sybil attacks via social networks. *ACM SIGCOMM Computer Communication Review*, 36(4):267–278, 2006.
- [59] Chao Michael Zhang and Vern Paxson. Detecting and analyzing automated activity on twitter. In *International Conference on Passive and Active Network Measurement*, pages 102–111. Springer, 2011.
- [60] Binbin Zhao, Haiqin Weng, Shouling Ji, Jianhai Chen, Ting Wang, Qinming He, and Reheem Beyah. Towards evaluating the security of real-world deployed image captchas. In *ACM Workshop on Artificial Intelligence and Security*, 2018.

SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub

Md Omar Faruk Rokon
UC Riverside
mroko001@ucr.edu

Risul Islam
UC Riverside
risla002@ucr.edu

Ahmad Darki
UC Riverside
adark001@ucr.edu

Evangelos E. Papalexakis
UC Riverside
epapalex@cs.ucr.edu

Michalis Faloutsos
UC Riverside
michalis@cs.ucr.edu

Abstract

Where can we find malware source code? This question is motivated by a real need: there is a dearth of malware source code, which impedes various types of security research. Our work is driven by the following insight: public archives, like GitHub, have a surprising number of malware repositories. Capitalizing on this opportunity, we propose, SourceFinder, a supervised-learning approach to identify repositories of malware source code efficiently. We evaluate and apply our approach using 97K repositories from GitHub. First, we show that our approach identifies malware repositories with 89% precision and 86% recall using a labeled dataset. Second, we use SourceFinder to identify 7504 malware source code repositories, which arguably constitutes the largest malware source code database. Finally, we study the fundamental properties and trends of the malware repositories and their authors. The number of such repositories appears to be growing by an order of magnitude every 4 years, and 18 malware authors seem to be "professionals" with a well-established online reputation. We argue that our approach and our large repository of malware source code can be a catalyst for research studies, which are currently not possible.

1 Introduction

Security research could greatly benefit by an extensive database of malware source code, which is currently unavailable. This is the assertion that motivates this work. First, security researchers can use malware source code to: (a) understand malware behavior and techniques, and (b) evaluate security methods and tools. In the latter, having the source code can provide the groundtruth for assessing the effectiveness of different techniques, such as reverse engineering methods. Second, currently, a *malware source code* database is not readily available. By contrast, there are several databases with *malware binary code*, as collected via honeypots, but even those are often limited in number and not widely available. We discuss existing malware archives in Section 9.

A missed opportunity: Surprisingly, software archives, like GitHub, host many publicly-accessible malware reposi-

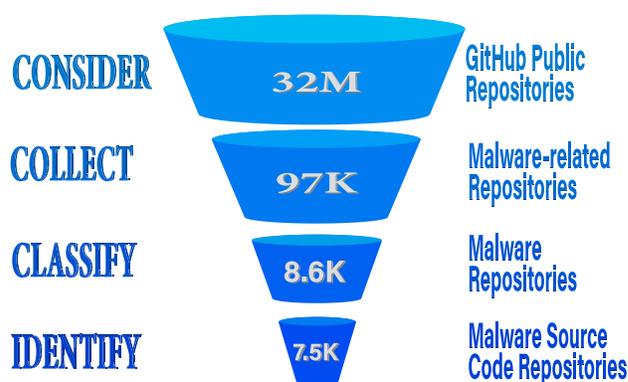


Figure 1: The steps of our work as a funnel: We identify 7.5K malware source code repositories in GitHub starting from 32M repositories based on 137 malware keywords (Q137).

tories, but this has not yet been explored to provide security researchers with malware source code. In this work, we focus on GitHub which is arguably the largest software storing and sharing platform. As of October 2019, GitHub reports more than 34 million users [25] and more than 32 million public repositories [24]. As we will see later, there are thousands of repositories that have malware source code, which seem to have escaped the radar of the research community so far. We use a broad definition of malware to include any repository containing software that can participate in compromising devices and supporting offensive, undesirable and parasitic activities.

Why do authors create public malware repositories? This question mystified us: these repositories expose both the creators and the intelligence behind the malware. Intrigued, we conducted a small investigation on malware authors, as we discuss below.

Problem: How can we find malware source code repositories in a large archive, like GitHub? The input to the problem is an online archive and the desired output is a database of

malware repositories. The challenges include: (a) collecting an appropriate set of repositories from the potentially vast archive, and (b) identifying the repositories that contain malware. Optionally, we also want to further help researchers that will potentially use these repositories, by determining additional properties, such as the most likely target platform, the malware type or family etc. Another practical challenge is the need to create the ground truth for validation purposes.

Related work: To the best of our knowledge, there does not seem to be any study focusing on the problem above. We group related works in the following categories. First, several studies analyze software repositories to find usage and limitations without any focus on malware [14]. Second, several efforts create and maintain databases of malware binaries but without source code [2, 3]. Third, many efforts attempt to extract higher-level information from binaries, such as lifting to Intermediate Representation (IR) [20], but it is really difficult to re-create the source code [10]. In fact, such studies would benefit from our malware source-code archive to evaluate and improve their methods. Taking a software engineering angle, an interesting work [8] compares the evolution of 150 malware source code repositories with that of benign software. We discuss related works in Section 9.

Contributions: Our work is arguably the first effort to systematically identify malware source code repositories from a massive public archive. The contribution of this work is three-fold: (a) we propose SourceFinder, a systematic approach to identify malware source-code repositories with high precision, (b) we create, arguably, the largest non-commercial malware source code archive with 7504 repositories, and (c) we study patterns and trends of the repository ecosystem including temporal and author-centric properties and behaviors. We apply and evaluate our method on the GitHub archive, though it could also be used on other archives, as we discuss in Section 8.

Our key results can be summarized in the following points, and some key numbers are shown in Figure 1.

a. We collect **97K malware-related repositories** from GitHub, namely repositories retrieved using malware keywords through GitHub's API and employing techniques to overcome several limitations. We also generate an extensive groundtruth with 2013 repositories, as we explain in Section 3.

b. **SourceFinder achieves 89% precision.** We systematically consider different Machine Learning approaches, and carefully-created representations for the different fields of the repository, such as title, description etc. We then systematically evaluate the effect of the different features, as we discuss in Section 5. We show that we classify malware repositories with a 89% precision, 86% recall and 87% F1-score using five fields from the repository.

c. **We identify 7504 malware source-code repositories**, which is arguably the largest malware source-code database available to the research community. We have already downloaded the contents in these repositories, in case GitHub de-

letes to deactivate them. We also create a curated database of 250 malware repositories, manually verified and spanning a wide range of malware types.

d. **The number of new malware repositories in our data more than triples every four years.** The increasing trend is interesting and alarming at the same time.

e. **We identify popular and influential repositories.** We study the malware repositories using three metrics of popularity: the number of watchers, forks and stars. We find 8 repositories that dominate the top-5 lists for all three metrics.

f. **We identify prolific and influential authors.** We find that 3% of the authors have more than 300 followers. We also find that 0.2% of the authors have more than 7 malware repositories, with the most prolific author *cyberthreats* having created 336 repositories.

g. **We identify and profile 18 professional hackers.** We find 18 authors of malware repositories, who seem to have created a brand around their activities, as they use the same user names in security forums. For example, user *3vilp4wn* (pronounced evil-pawn) is the author of a keylogger malware in GitHub, which the author is promoting in the *Hack This Site* forum using the same username. We present our study of malware authors in Section 7.

Open-sourcing for maximal impact: creating an engaged community. We intend to make our datasets and our tools available for research purposes at our website [28]. Our vision is to create community-driven reference platform, which will provide: (a) malware source code repositories, (b) community-vetted labels and feedback, and (c) open-source tools for collecting and analyzing malware repositories. Our goal is to expand our database with more software archives and richer information. Although authors could start hiding their repositories (see Section 8), we argue that our already-retrieved database could have significant impact in enabling certain types of security studies [22, 29, 32].

2 Background

We provide background information on GitHub and the type of information that repositories have.

GitHub is a massive world-wide software archive, which enables users to share code through its public repositories thus creating a global social network of interaction. For instance, first, users can collaborate on a repository. Second, users often "fork" projects: they copy and evolve projects. Third, users can follow projects, and "up-vote" projects using "stars" (think Facebook likes). Although GitHub has many private repositories, there are 32 million public software repositories.

We describe the key elements of a GitHub repository. A repository is equivalent to a project folder, and typically, each repository corresponds to a single software project. However, a repository could contain: (a) source code, (b) binary code, (c) data, (d) documents, such as latex files, and (e) all of the above.

A repository in GitHub has the following data fields: a) title,

b) description, c) topics, d) README file, e) file and folders, f) date of creation and last modified, g) forks, h) watchers, i) stars, and j) followers and followings, which we explain below.

a. Repository title: The title is a mandatory field and it usually consists of less than 3 words.

b. Repository description: This is an optional field that describes the objective of the project and it is usually 1-2 sentences long.

c. Repository topics: An author can optionally provide topics for her repository, in the form of tags, for example, "*linux, malware, malware-analysis, anti-virus*". Note that 97% of the repositories in our dataset have less than 8 topics.

d. README file: As expected, the README file is a documentation and/or light manual for the repository. This field is optional and its size varies from one or two sentences to many paragraphs. For example, we found that 17.48% of the README files in our repositories are empty.

e. File and folders: In a well-constructed software, the file and folder names of the source code can provide useful information. For example, some malware repositories contain files or folders with indicative names, such as "malware", "source code" or even specific malware types or names of specific malware, like *mirai*.

f. Date of creation and last modification: GitHub maintains the date of creation and last modification of a repository. We find malware repository created in 2008 are actively being modified by authors till present.

g. Number of forks: Users can fork a public repository: they can create a clone of the project. An user can fork any public repository to change locally and contribute to the original project if the owner accepts the modification. The number of forks is an indication of the popularity and impact of a repository. Note that the number of forks indicates the number of distinct users that have forked a repository.

h. Number of watchers: Watching a repository is equivalent to "following" in the social media language. A "watcher" will get notifications, if there is any new activity in that project. The numbers of watchers is an indication of the popularity of a repository [16].

i. Number of stars: A user can "star" a repository, which is equivalent to the "like" function in social media [5], and places the repository in the users favorite group, but does not provide constant updates as with the "watching" function.

j. Followers: Users can also follow other users' work. If A follows B, A will be added to B's followers and B will be added to A's following list. The number of followers is an indication of the popularity of a user [39].

3 Data Collection

The first step in our work is to collect repositories from GitHub that have a higher chance of being related to malware. Extracting repositories at scale from GitHub hides several

Set	Descriptions	Size
Q1	Query set = {"malware"}	1
Q50	Query with 50 keywords with $Q1 \subset Q50$	50
Q137	Query with 137 keywords with $Q50 \subset Q137$	137
RD1	Retrieved repositories from query Q1	2775
RD50	Retrieved repositories from query Q50	14332
RD137	Retrieved repositories from query Q137	97375
LD1	Labeled subset of RD1 dataset	379
LD50	Labeled subset of RD50 dataset	755
LD137	Labeled subset of RD137 dataset	879
M1	Malware source code repositories in RD1	680
M50	Malware source code repositories in RD50	3096
M137	Malware source code repositories in RD137	7504
MCur	Manually verified malware source code dataset	250

Table 1: Datasets, their relationships, and their size.

subtleties and challenges, which we discuss below.

Using the GitHub Search API, a user can query with a set of keywords and obtain the most relevant repositories. We describe briefly how we select appropriate keywords, retrieve related repositories from GitHub and how we establish our ground truth.

A. Selecting keywords for querying: In this step, we want to retrieve repositories from GitHub in a way that: (a) provides as many as possible malware repositories, and (b) provides a wide coverage over different types of malware. For this reason, we select keywords from three categories: (a) malware and security related keywords, such as malware and virus, (b) malware type names, such as ransomware and keylogger, and (c) popular malware names, such as mirai. Due to space limitations, we will provide the full list of keywords in our website at publication time for repeatability purposes.

We define three sets of keywords that we use to query GitHub. The reason is that we want to assess the sensitivity of the number of keywords on the outcome. Specifically, we use the following query sets: (a) the **Q1 set**, which only contains the keyword "malware"; (b) the **Q50 set**, which contains 50 keywords, and (c) the **Q137 set** which contains 137 keywords. The Q137 keyword set is a super-set of Q50, and Q50 is a superset of Q1. As we will see below, using the query set Q137 provides wider coverage, and we recommend in practice. We use the other two to assess the sensitivity of the results in the initial set of keywords. We list our datasets in Table 1.

B. Retrieving related repositories: Using the Search API, we query GitHub with our set of keywords. Specifically, we query GitHub with every keyword in our set separately. In an ideal world, this would have been enough to collect all related repositories: a query with "malware" (Q1) should return the many thousands related repositories, but this is not the case.

The search capability hides several subtleties and limitations. First, there is a limit of 1000 repositories that a single

Labeled Dataset	Malware Repo.	Benign Repo.
LD137	313	566
LD50	326	429
LD1	186	193

Table 2: Our groundtruth: labeled datasets for each of the three queries, for a total of 2013 repositories.

search can return: we get the top 1000 repositories ordered by relevancy to the query. Second, the GitHub API allows 30 requests per minute for an authenticated user and 10 requests per minute for an unauthenticated user.

Bypassing the API limitations. We were able to find a work around for the first limitation by using ranking option. Namely, a user can specify her preferred ranking order for the results based on: (a) best match, (b) most stars, (c) fewest stars, (d) most forks, (e) fewest forks, (f) most recently updated, and (g) the least recently updated order. By repeating a query with all these seven ranking options, we can maximize the number of distinct repositories that we get. This way, for each keyword in our set, we search with these seven different ranking preferences to obtain a list of GitHub repositories.

C. Collecting the repositories: We download all the repositories identified in our queries using PyGithub [52], and we obtain three sets of repositories RD1, RD50 and RD137. These retrieved datasets have the same "subset" relationship that they query sets have: $RD1 \subset RD50 \subset RD137$. Note that we remove pathological repositories, mainly repositories with no actual content, or repositories "deleted" by GitHub. For each repository, we collect and store: (a) repository-specific information, (b) author-specific information, and (c) all the code within the repository.

As we see from Table 1, using more and specialized malware keywords returns significantly more repositories. Namely, searching with the keyword "malware" does return 2775 repositories, but searching with the Q50 and Q137 returns 14332 and 97375 repositories respectively.

D. Establishing the groundtruth: As there was no available groundtruth, we needed to establish our own. As this is a fairly technical task, we opted for domain experts instead of Mechanical Turk users, as recommended by recent studies [23]. We use three computer scientists to manually label 1000 repositories, which we selected in a uniformly random fashion, from each of our dataset RD137 and RD50 and 600 repositories from RD1. The judges were instructed to independently investigate every repository thoroughly.

Ensuring the quality of the groundtruth. To increase the reliability of our groundtruth, we took the following measures. First, we asked judges to label a repository *only*, if they were certain that it is malicious or benign and distinct, and leave it unlabeled otherwise. We only kept the repositories for which the judges agreed unanimously. Second, duplicate repositories were removed via manual inspection, and were excluded from the final labeled dataset to avoid overfitting. It is worth noting

that we only found very few duplicates in the order of 3-5 in each dataset with hundreds of repositories.

With this process, we establish three separate labeled datasets named LD137, LD50, and LD1 starting from the respective malware repositories from each of our queries, as shown in Table 2. Although the labeled datasets are not 50-50, they are representing both classes reasonably well, so that a naive solution that will label everything as one class, would perform poorly. By contrast, our approach performs sufficiently well, as we will see in Section 5.

As there is no available dataset, we argue that we make a sufficient size dataset by manual effort.

4 Overview of our Identification Approach

Here, we describe our supervised learning algorithm to identify the repositories that contain malware.

Step 1. Data preprocessing: As in any Natural Language Processing (NLP) method, we start with some initial processing of the text to improve the effectiveness of the solution. We briefly outline three levels of processing functionality.

a. Character level preprocessing: We handle the character level "noise" by removing special characters, such as punctuation and currency symbols, and fix Unicode and other encoding issues.

b. Word level preprocessing: We eliminate or aggregate words following the best practices of Natural Language Processing [33]. First, we remove article words and other words that don't carry significant meaning on their own. Second, we use a stemming technique to handle inflected words. Namely, we want to decrease the dimensionality of the data by grouping words with the same "root". For example, we group the words "organizing", "organized", "organize" and "organizes" to one word "organize". Third, we filter out common file and folder names that we do not expect to help in our classification, such as "LEGAL", "LICENSE", "gitattributes" etc.

c. Entity level filtering: We filter entities that are likely not helpful in describing the scope of a repository. Specifically, we remove numbers, URLs, and emails, which are often found in the text. We found that this filtering improved the classification performance. In the future, we could consider mining URLs and other information, such as names of people, companies or youtube channels, to identify authors, verify intention, and find more malware activities.

Step 2. The repository fields: We consider fields from the repositories that can be numbers or text. Text-based fields require processing in order to turn them into classification features and we explain this below. We use and evaluate the following text fields: title, description, topics, file and folder names and README file fields.

Text field representation: We consider two techniques to represent each text field by a feature in the classification.

a. Bag of Words (BoW): The bag-of-words (BoW) model is among the most widely used representations of a document. The document is represented as the number of occurrences of

its words, disregarding grammar and word order [75]. This model is commonly used in document classification where the frequency of each word is used as feature value for training a classifier [42]. We use the model with the count vectorizer and TF-IDF vectorizer to create the feature vector.

In more detail, we represent each text field in the repository with a vector $V[K]$, where $V[i]$ corresponds to the significance of word i for the text. There are several ways to assign values $V[i]$: (a) zero-one to account for presence, (b) number of occurrences, and (c) the TF-IDF value of the word. We evaluated all the above methods.

Fixing the number of words per field. To improve the effectiveness of our approach using BoW, we conduct a feature selection process, χ^2 statistic following best practices [55]. The χ^2 statistic measures the lack of independence between a word (feature) and a class. A feature with lower chi-square score is less informative for that class, and thus not useful in the classification. We discuss this further in Section 5. For each text-based field f , we select the top K_f words for that field, which exhibit the highest discerning power in identifying malware repositories. Note that we set a value for K_f during the training stage. For each field, we select the value K_f , as we explain in Section 5.

b. Word embedding: The word embedding model is a vector representations of each word in a document: each word is mapped to an M -dimensional vector of real numbers [44], or equivalently are projected in an M -dimensional space. A good embedding ensures that words that are close in meaning have nearby representations in the embedded space. In order to create the document vector, word embedding follows two approaches (i) frequency-based vectorizer(unsupervised) [58] and (ii) content-based vectorizer(supervised) [38]. Note that in this type of representation, we do not use the *word level processing*, which we described in the previous step, since this method can leverage contextual information.

We use frequency-based word embedding with word average and TF-IDF vectorizer. We also use pre-trained model of Google word2vec [43] and Stanford (Glov) [49] to create the feature vector.

Finally, we create the vector of the repository by concatenating the vectors of each field of that repository.

Step 3. Selecting the fields: Another key question is which fields from the repository to use in our classification. We experiment with all of the fields listed in Section 2 and we explain our findings in the next Section.

Step 4. Selecting a ML engine: We design ML model to classify the repositories into two classes: (i) malware repository and (ii) benign repository. We systematically evaluate many machine learning algorithms [7, 45]: Naive Bayes (NB), Logistic Regression (LR), Decision Tree (CART), Random Forest(RF), K-Nearest Neighbor (KNN), Linear Discriminant Analysis (LDA), and Support Vector Machine (SVM).

Step 5. Detecting source code repositories: In this final step, we want to identify the presence of source code in the

repositories. By June 2020, GitHub started labeling repositories that contain source code. Therefore, one can simply filter out all repositories that are not labelled as such.

As our study predates this GitHub feature, we developed a heuristic approach to identify source code repositories independently, which we describe below. Our heuristic exhibits 100% precision as validated by GitHub's classification, as we will see in Section 5.

Our source-code classification heuristics works in two steps. First, we identify files in the repository that contain source code. To do this, we start by examining their file extension. If the file extension is one of the known programming languages: *Assembly, C, C++, Batch File, Bash Shell Script, Power Shell Script, Java, Python, C#, Objective-C, Pascal, Visual Basic, Matlab, PHP, Javascript, and Go*, we label it as a source file. Second, if the number of source files in a repository exceeds the **Source Percentage threshold (SourceThresh)**, we consider that the repository contains source code.

5 Evaluation: Choices and Results

In this section, we evaluate the effectiveness of the classification based on the proposed methodology defined in Section 4. More specifically, our goal here is to answer the following questions:

1. **Repository field selection:** Which repository fields should we consider in our analysis?
2. **Field representation:** Which feature representation is better between bag of words (BoW) and word embedding and considering several versions of each?
3. **Feature selection:** What are the most informative features in identifying malware repositories?
4. **ML algorithm selection:** Which ML algorithm exhibits the best performance?
5. **Classification effectiveness:** What is the precision, recall and F1-score of the classification?
6. **Identifying malware repositories:** How many malware repositories do we find?
7. **Identifying malware source code repository:** How many of the malware repositories have source code?

Note that we have a fairly complex task: we want to identify the best fields, representation method and Machine Learning engine, while considering different values for parameters. What complicates matters is that all these selections are interdependent. We present our analysis in sequence, but we followed many trial and error and non-linear paths in reality.

1. Selecting repository fields: We evaluated all the repository fields mentioned earlier. In fact, we used a significant number of experiments with different subsets of the features, not shown here due to space limitations. We find that the title, description, topics, README file, and file and folder names have the most discerning power. We also considered number of forks, watchers, and stars of the repository and the number

Representation	Classification Accuracy Range
Bag of Words with Count Vectorizer	86%-51%
Bag of Words with Count Vectorizer + Feature Selection	91%-56%
Bag of Words with TF-IDF vectorizer	82%-63%
Word Embedding with Word Average	85%-72%
Word Embedding with TF-IDF	85%-74%
Pretrained Google word2vec Model	76%-64%
Pretrained Stanford (Glov) Model	73%-62%

Table 3: Selecting the feature representation model: We evaluate all the representations across seven machine learning approaches and report the range of the overall accuracy.

of followers and followings of the author of the repository. We found that not only it did not help, but it usually decreased the classification accuracy by 2-3%. One possible explanation is that the numbers of forks, stars and followers reflect the popularity rather than the content of a repository.

2. Selecting a field representation: The goal is to find, which representation approach works better. In Table 3, we show the comparison of the range of classification accuracy across the 7 different ML algorithms that we will also consider below. We find that Bag of Words with the count vectorizer representation reaches 86% classification accuracy, with the word embedding approach nearly matching that with 85% accuracy. Note that we finetune the selection of words to represent each field in the next step.

Why does not the embedding approach outperform the bag of words? One would have expected that the most complex embedding approach would have been the winner and by a significant margin. We attribute this to the relatively small text size in most text fields, which also do not provide well-structured sentences (think two-three words for the title, and isolated words for the topics). Furthermore, the word co-occurrences does not exist in the topics and file names fields, which is partly what makes embedding approaches work well in large and well structured documents [26, 41].

In the rest of this paper, we use the Bag of Words with count vectorizer to represent our text fields, since it exhibits good performance and is computationally less intensive than the embedding method.

3. Fixing the number of words per field. We want to identify the most discerning words from each text field, which is a standard process in NLP for improving the scalability, efficiency and accuracy of a text classifier [12]. Using the χ^2 statistic, we select the top K_f best words from each field.

To select the appropriate number of words per field, we followed the process below. We vary $K_f = 5, 10, 20, 30, 40$ and 50 for title, topic and README file, and we find that the top 30 words in title, 10 words in topic and 10 words in README file exhibit the highest accuracy. Similarly, we try $K_f = 80, 90, 100, 110$ and 120 for file names and $K_f = 300, 325, 350,$

375, 400, 425, 450 and 475 for the description field. We find that the top 100 words for file and folder names and top 400 words for description field give the highest accuracy. Note that we do this during training and refining the algorithm, and then we continue to use these words as features in testing.

Thus, we select the top: (a) 30 words from the title, (b) 10 words from the topics, (c) 400 words from the description, (d) 100 words from the file names, and 10 words from the README file. This leads to a total of 550 words across all fields. For reference, we find 9253 unique words in the repository fields of our training dataset. Reducing the focus on the top 550 most discerning words per field increases the classification accuracy by as much as 20% in some cases.

4. Evaluating and selecting ML algorithms: We find that Multinomial Naive Bayes exhibits the best F1-score with 87%, striking a good balance between 89% precision and 86% recall for the malware class among other machine learning classifier which we considered. Detecting the benign class, we do even better with 92% precision, 94% recall and 93% F1-score. By contrast, the F1-score of the other algorithms is below 79%. Note that KNN, LR and LDA methods provide higher precision, but with significantly lower recall. Thus, one could use these algorithms to get higher precision at the cost of lower total number of repositories.

We use Multinomial Naive Bayes as our classification engine for the rest of this study. We attempt to explain the superior F1-Score of the Naive Bayes in our context. The main advantage of Naive Bayes over other algorithms is that it considers the features independently of each other for a given class and can handle large number of features better. As a result, it is more robust to noisy or unreliable features. It also performs well in domains with many equally important features, where other approaches suffer, especially with a small training data, and it is not prone to overfitting [64]. As a result, the Naive Bayes is considered a dependable algorithm for text classification and it is often used as the benchmark to beat [71].

5. Assessing the effect of the query set: We have made the following choices in the previous steps: (a) 5 text-based fields, (b) bag of words with count vectorization, (c) 550 total words across all the fields, and (d) the Multinomial Naive Bayes. We perform 10-fold cross validation and report the precision, recall and F1-score in Figure 2 for our three different labeled data sets. We see that the precision stays above 89% for all three datasets, with a recall above 77%.

It is worth noting the relative stability of our approach with respect to the keyword set for the initial query especially between LD50 and LD137 datasets. The LD1 dataset we observe higher accuracy, but significantly less recall compared to LD137. We attribute this fact to the single keyword used in selecting the repositories in LD1, which may have lead to a more homogeneous group of repositories. Interestingly, LD50 seem to have the lower recall and F1-score even though the differences are not that large.

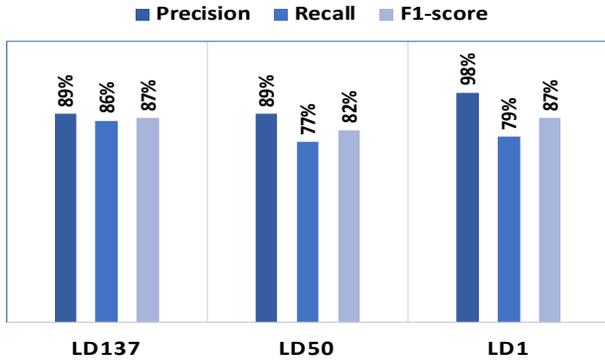


Figure 2: Assessing the effect of the number of keywords in the query: Precision, Recall and F1-score of our approach on the LD137, LD50 and LD1 labeled datasets.

Dataset	Initial	Malware	Mal. + Source
RD1	2775	809	680
RD50	14332	3615	3096
RD137	97375	8644	7504

Table 4: The identified repositories per dataset with: (a) malware, and (b) malware and source code.

6. Identifying 8644 malware repositories: We use LD137 to train our Multinomial Naive Bayes model and apply it on RD137 dataset. We find 8644 malware repositories. We also apply the same trained model on RD1 and RD50 and find 809 and 3615 malware repositories respectively, but this repositories are included in the 8644. (Recall that RD1 and RD50 are subsets of RD137).

7. Identifying 7504 malware source code repositories: As of June 2020, we can use the source code labelling to identify such repositories. Here, we use this labelling to validate our heuristic approach for completeness.

In deploying our heuristic, we set our Source Percentage threshold to 75%, meaning that: if more than 75% of files in a repository are source code files, we label it as a source code repository. Applying this heuristic, we find that 7504 repositories are most likely source code repositories in RD137. We use the name **M137** to refer to this group of malware source code repositories. We find 680 and 3096 malware source code repositories in RD1 and RD50 as shown in Table 4. However, these are subset of M137, given that RD1 and RD50 are subsets of RD137.

We find that 100% of our source code repositories are also labeled as such by GitHub. We argue that our heuristic could be useful for other software archives, which may not provide the "source code" label.

8. A curated malware source code dataset: MCur As a tangible contribution, we provide, MCur, a dataset of 250 repositories from the M137 dataset, which we manually verify for containing malware source code and relating to a partic-

ular malware type. Opting for diversity and coverage, the dataset spans all the identified types: virus, backdoor, botnet, keylogger, worm, ransomware, rootkit, trojan, spyware, spoof, ddos, sniff, spam, and cryptominer. We intend to constantly update and make our labeled malware repositories publicly available [28].

6 A large scale study of malware

Encouraged by the substantial number of malware repositories, we study the distributions and longitudinal properties of the identified malware repositories in M137.

Caveat: We provide some key observations in this section, but they should be viewed as indicative and approximate trends and only within the context of the collected repositories and with the general assumption that repository titles and descriptions are reasonably accurate. In Section 8, we discuss issues around the biases and limitations that our dataset may introduce.

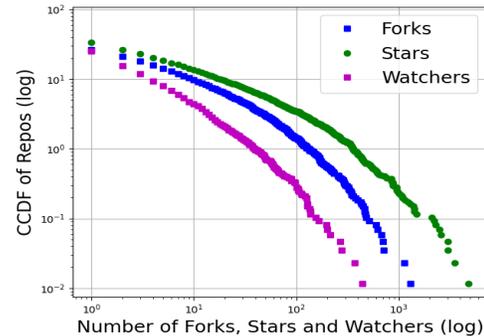


Figure 3: CCDF distributions of forks, stars and watchers per repository.

A. Identifying influential repositories. The prominence of a repository can be measured by the number of *forks*, *stars*, and *watchers*. In Figure 3, we plot the complementary cumulative distribution function (CCDF) of these three metrics for our malware repositories.

Fork distribution: We find that 2% of the repositories seem quite influential with at least 100 forks as shown in Figure 3. Recall that the fork counter indicates the number of distinct users that have forked a repository. For reference, 78% of the repositories have less than 2 forks.

Star distribution: We find that 2% of the repositories receive more than 250 stars as shown in Figure 3. For reference, 75% of the repositories have less than 3 stars.

Watcher distribution: In Figure 3, we find that 1% of the repositories have more than 50 watchers. For reference, we observe that 84% of the repositories have less than 3 watchers. Note that these distributions are skewed, and follow patterns that can be approximated by a log-normal distribution.

Which are the most influential repositories? We find that 8 repositories dominate the top 5 spots across all three metrics:

R ID	Author	# Star	# Fork	# Watcher	Content of the Repository
1	ytisf	4851	1393	730	80 malware source code and 140 Binaries
2	n1nj4sec	4811	1307	440	Pupy RAT
3	Screetsec	3010	1135	380	TheFatRat Backdoor
4	malwardllc	2515	513	268	Byob botnet
5	RoganDawes	2515	513	268	USB attack platform
6	Visgean	626	599	127	Zeus trojan horse
7	Ramadhan	535	283	22	30 malware samples
8	dana-at-cp	1320	513	125	backdoor-apk backdoor

Table 5: The profile of the top 5 most influential malware repositories across all three metrics with 8 unique repositories.

stars, forks, and watchers. We present a short profile of these dominant repositories in Table 5. Most of the repositories contain a single malware project, which is an established practice among the authors in GitHub [48,66]. We find that the repository "theZoo" [72], created by *ytisf* in 2014 is the most forked, watched, and starred repository with 1393 forks, 730 watchers and 4851 stars as of October, 2019. However, this repository is quite unique and was created with the intention of being a malware database with 140 binaries and 80 source code repositories.

Influence metrics are correlated: As one would expect, the influence and popularity metrics are correlated. We use a common correlation metric, the Pearson Correlation Coefficient (r) [6], measured in a scale of $[-1, 1]$. We calculate the metric for all pairs of our three popularity metrics. We find that all of them exhibit higher positive correlation: stars vs. forks ($r = 0.92$, $p < 0.01$), forks vs. watchers ($r = 0.91$, $p < 0.01$) and watchers vs. stars ($r = 0.91$, $p < 0.01$).

B. Malware type and target platform. We wanted to get a feel for what type of malware we have identified. As a first approximation, we use the keywords found in the text fields to relate repositories in M137 with the type of malware and the intended target platform. Our goal is to create the two-dimensional distribution per malware type and the target platform as shown in Table 6. To create this table, we associate a repository with keywords in its title, topics, descriptions, file names and README file fields of: (a) the 6 target platforms, and (b) the 13 malware type keywords.

How well does this heuristic approach work? We provide two different indications of its relative effectiveness. First, the vast majority of the repositories relate to one platform or type of malware: (a) less than 8% relate to more than one platform, and (b) less than 11% relate to more than one type of malware. Second, we manually verify the 250 repositories in our curated data MCur and find a 98% accuracy.

Below, we provide some observations from Table 6.

a. Keyloggers reign supreme. We see that one of the largest categories is the keylogger malware with 679 repositories, which are mostly affiliated with Windows and Linux platforms. We discuss the emergence of keyloggers below in

Types	Target Platform						Total
	Wind.	Linux	Mac	IoT	Andr.	iOS	
Total	1592	1365	380	108	442	131	4018
key-logger	396	209	42	2	27	3	679
back-door	181	227	37	11	51	4	511
virus	235	131	34	2	51	16	469
botnet	153	154	43	36	64	17	467
trojan	133	70	24	16	67	19	329
spooof	76	115	88	2	20	9	310
rootkit	55	163	13	2	19	3	255
ransom-ware	117	67	14	1	33	13	245
ddos	71	95	20	10	9	3	208
worm	61	45	18	5	25	18	172
spyware	45	22	6	6	38	16	133
spam	40	29	18	14	23	5	129
sniff	29	38	23	1	15	5	111

Table 6: Distribution of the malware repositories from M137 dataset based on the malware type and malware target platform. This table demonstrates the repositories that fit with the criteria defined in Section 6.

our temporal analysis.

b. Windows and Linux are the most popular targets. Not surprisingly, we find that the majority of the malware repositories are affiliated with these two platforms: 1592 repositories for Windows, and 1365 for Linux.

c. MacOS-focused repositories: fewer, but they exist. Although MacOS platforms are less commonly targeted, we find a non-trivial number of malware repositories for MacOS. As shown in Figure 4c, there are 380 MacOS malware repositories, which is roughly an order of magnitude less compared to those for Windows and Linux.

C. Temporal analysis. We want to study the evolution and the trends of malware repositories. We plot the number of new malware repositories per year: a) total malware, b) per type of malware, and c) per target platform in Figure 4. We discuss a few interesting temporal behaviors below.

a. The number of new malware repositories more than triples every four years. We see an alarming increase from 117 malware repositories in 2010 to 620 repositories in 2014 and to 2166 repositories in 2018. We also observe a sharp increase of 70% between 2015 to 2016 shown in Figure 4a.

b. Keyloggers started a super-linear growth since 2010 and are by far affiliated with the most new repositories per year since 2013, but their rate of growth reduced in 2018.

c. Ransomware repositories emerge in 2014 and gain momentum in 2017. Ransomware experienced their highest growth rate in 2017 with 155 new repositories, while that number dropped to 103 in 2018.

d. Malware activity slowed down in 2018 across the board. It seems that 2018 is a slower year for all malware even when seen by type (Figure 4b) and target platform (Fig-

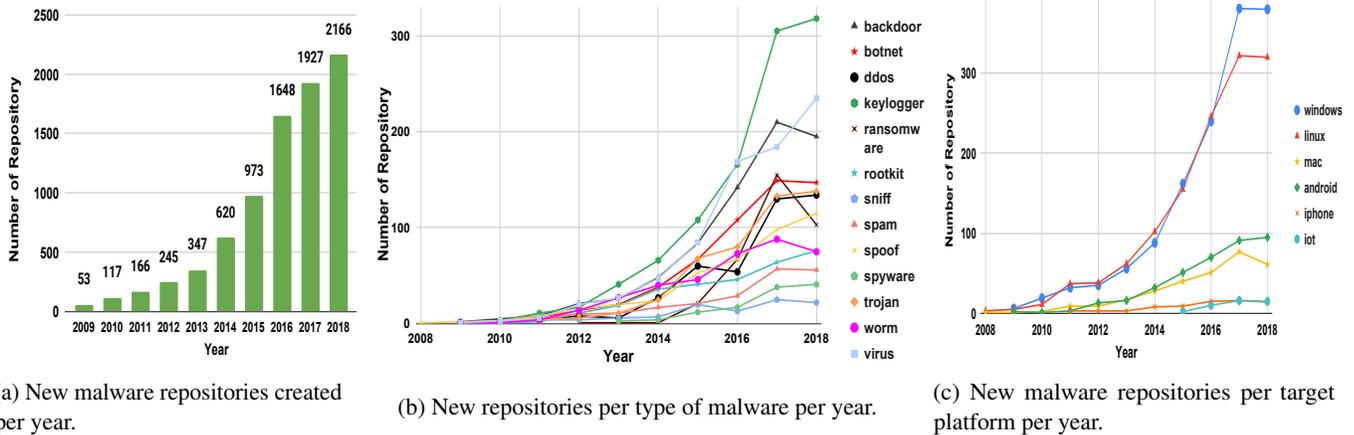


Figure 4: New malware repositories per year: a) all malware, b) per type of malware, and c) per target platform.

ure 4c). We find that the number of new malware repositories has dropped significantly in 2018 for most types of malware except virus, keylogger and trojan.

e. IoT and iPhone malware repositories become more visible after 2014. We find that IoT malware emerges in 2015 and iPhone malware sees an increase after 2014 in Figure 4c. We conjecture that this is possibly encouraged by the emergence and increasing popularity of specific malware: (a) WireLurker, Masque, AppBuyer malware [13] for iPhones, and (b) BASHLITE [70], a Linux based botnet for IoT devices. We find the names of the aforementioned malware in many repositories starting in 2014. Interestingly, the source code of the original BASHLITE botnet is available in a repository created by *anthonygtellez* in 2015.

f. Windows and Linux: dominant but slowing down. In Figure 4c, we see that windows and linux malware are flattened between 2017 and 2018. By contrast, IoT and android repositories have increased.

7 Understanding malware authors

Intrigued by the fact that authors create public malware repositories, we attempt to understand and profile their behavior.

As a first step towards understanding the malware authors, we want to assess their popularity and influence. We use the following metrics: (a) number of malware repositories which they created, (b) number of followers, (c) total number of watchers on their repositories, and (d) total number of stars. We focus on the first two metrics here. We use the notation *top k authors* for any of the metrics above, where *k* can be any positive integer to referring to "heavy-hitters".

A. Finding influential malware authors. We study the distribution of the number of malware repositories created and the number followers per author in following.

First, we find that 15 authors are contributing roughly 5% of all malware repositories by examining the CCDF of the created repositories in Figure 5. From the figure, we find an

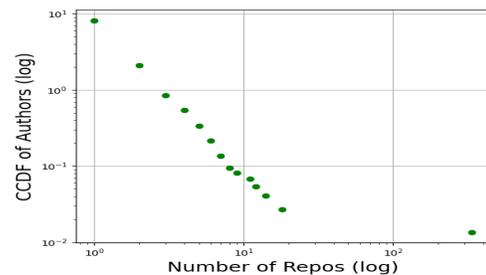


Figure 5: CCDF of malware repositories per author.

outlier author, *cyberthreats*, who doesn't follow power law distribution [21], has created 336 malware repositories. We also find that 99% authors have less than 5 repositories.

Second, we study the distribution of the number of followers per author, but omit the plot due to space limitations. The distributions is skewed with 3% (221) of the authors having more than 300 followers each, while 70% of the authors have less than 16 followers.

B. Malware authors strive for an online "brand": In an effort to understand the motive of sharing malware repositories, we make the following investigation.

a. Usernames seem persistent across online platforms. We find that many malware authors use the same username consistently across many online platforms, such as security forums. We conjecture that they are developing a reputation and they use their username as a "unique" identifier.

We identify 18 malware authors¹, who are active in at least one of the three security forums: Offensive Community, Ethical Hacker and Hack This Site, for which we happen to have access to their data. We conjecture that at least some of these usernames correspond to the same users based on the follow-

¹Note that this does not mean that the other authors are not doing the same, but they could be active in other security forums or online platforms.

ing two indications. First, we find direct connections between the usernames across different platforms. For example, user *3vilp4wn* at the "Hack This Site" forum is promoting a keylogger malware by referring to a GitHub repository [1] whose author has the same username. Second, these usernames are fairly uncommon, which increases the likelihood of belonging to the same person. For example, there is a GitHub user with the name *fahimmagsi*, and someone with the same username is boasting about their hacking successes in the "Ethical Hacker" forum. As we will see below, *fahimmagsi* seems to have a well-established online reputation.

b. "Googling" usernames reveals significant hacking activities. Given that these GitHub usernames are fairly unique, it was natural to look them up on the web at large. Even a simple Internet search with the usernames reveals significant hacking activities, including hacking websites or social networks, and offering hacking tutorials in YouTube.

We investigate the *top 40* most prolific malware authors using a web search with a *single* simple query: "hacked by <username>". We then examine only the first page of search results. Despite all these self-imposed restrictions, we identify three users with substantial hacking related activities across Internet. For example, we find a number of news articles for hacking a series of websites by GitHub users *fahimmagsi* and *CR4SH* [65] [15]. Moreover, we find user *nInj4sec* sharing a multi-functional Remote Access Trojan (RAT) named "Pupy", developed by her, which received significant news coverage in security articles back in March of 2019 [46] [54]. We are confident that well-crafted and targeted searches can connect more malware authors with hacking activities and usernames in other online forums.

8 Discussion

We discuss the effectiveness and limitations of SourceFinder.

a. Why is malware publicly available in the first place? Our investigation in Section 7 provides strong indications that malware authors want to actively establish their hacking reputation. It seems that they want to boost their online credibility, which often translates to money. Recent works [18, 51, 57] study the underground markets of malware services and tools: it stands to reason that notorious hackers will attract more clients. At the same time, GitHub acts as a collaboration platform, which can help hackers improve their tools.

b. Do we identify every malware repository in GitHub? Our tool can not guarantee that it will identify every malware repository in GitHub. First, we can only identify repositories that "want to be found": (a) they must be public, and (b) they must be described with the appropriate text and keywords. Clearly, if the author wants to hide her repository, we won't be able to find it. However, we argue that this defeats the purpose of having a public archive: if secrecy was desired, the code would have been shared through private links and services. Second, our approach is constrained by GitHub querying limitations, which we discussed in Section 3, and the

set of 137 keywords that we use. However, we are encouraged by the number and the reasonable diversity of the retrieved repositories we see in Table 6.

c. Are our datasets representative? This is the typical hard question for any measurement or data collection study. First of all, we want to clarify that our goal is to create a large database of malware source code. So, in that regard, we claim that we accomplished our mission. At the same time, we seem to have a fair number of malware samples in each category of interest, as we see in Table 6.

Studying the trends of malware is a distant second goal, which we present with the appropriate caveat. On the one hand, we are limited by GitHub's API operation, as we discussed earlier. On the other hand, we attempt to reduce the biases that are under our control. To ensure some diversity among our malware, we added as many words as we could in our 137 malware, which is likely to capture a wide range of malware types. We argue that the fairly wide breadth of malware types in Table 6 is a good indication. Note that our curated dataset MCur with 250 malware is reasonably representative in terms of coverage.

d. What is the overlap among the identified repositories? Note that our repository does not include forked repositories, since **GitHub does not return forked repositories as answers to a query**. Similarly, the breadth of the types of the malware as shown in Table 6 hints at a reasonable diversity. However, our tool cannot claim that the identified repositories are distinct nor is it attempting to do so. GitHub does not restrict authors from copying (downloading), and uploading it as a new repository. In the future, we intend to study the similarity and evolution among these repositories.

e. Are the authors of repositories the original creator of the source code? This is an interesting and complex question that goes beyond the scope of this work. Identifying the original creator will require studying the source code of all related repositories, and analyzing the dynamics of the hacker authors, which we intend to do in the future.

f. Are all the malware authors malicious? Not necessarily. This is an interesting question, but it is not central to the main point of our work. On the one hand, we find some white hackers or researchers, such as Yuval Nativ [74], or Nicolas Verdier [47]. On the other hand, several authors seem to be malicious, as we saw in Section 7.

g. Are our malware repositories in "working order"? It is hard to know for sure, but we attempt to answer indirectly. First, we pick 30 malware source codes and all of them compiled and a subset of 15 of them actually run successfully in an emulated environment as we already mentioned. Second, these public repositories are a showcase for the skills of the author, who will be reluctant to have repositories of low quality. Third, public repositories, especially popular ones, are inevitably scrutinized by their followers.

h. Can we handle evasion efforts? Our goal is to create the largest malware source-code database possible and having

collected 7504 malware repositories seems like a great start. In the future, malware authors could obfuscate their repositories by using misleading titles, and description, and even filenames. We argue that authors seem to want their repositories to be found, which is why they are public. We also have to be clear: it is easy for the authors to hide their repositories, and they could start by making them private or avoid GitHub altogether. However, both these moves will diminish the visibility and "street-cred" of the authors.

i. Will our approach generalize to other archives? We believe that SourceFinder can generalize to other archives, which provide public repositories, like GitLab and BitBucket. We find that these sites allow public repositories and let the users retrieve repositories. We have also seen equivalent data fields (title, description, etc). Therefore, we are confident that our approach can work with other archives.

9 Related Work

There are several works that attempt to determine if a piece of software is malware, usually focusing on a binary, using static or dynamic analysis [4, 17, 36, 60]. However, to the best of our knowledge, no previous study has focused on identifying malware source code in public software archives, such as GitHub, in a systematic manner as we do in this work. We highlight the related works in the following categories:

a. Studies that need malware source code. Several studies [40, 62, 78] use malware source code that are manually retrieved from GitHub repositories. Some studies [8] [9] compare the evolution and the code reuse of 150 malware source codes (with only some from GitHub) with that of benign software from a software engineering perspective and study the code reuse. Overall, various studies [22, 32] can benefit from malware source code to fine-tune their approach.

b. Mining and analyzing GitHub: Many studies have analyzed different aspects of GitHub, but not with the intention of retrieving malware repositories. First, there are efforts that study the user interactions and collaborations on GitHub and their relationship to other social media in [30, 37, 50]. Second, some efforts discuss the challenges in extracting and analyzing data from GitHub with respect to sampling biases [14, 27]. Other works [34, 35] study how users utilize the various features and functions of GitHub. Several studies [31, 53, 67] discuss the challenges of mining software archives, like *SourceForge* and GitHub, arguing that more information is required to make assertions about users and software projects. Finally, some efforts [61, 63, 76, 77] study GitHub repositories, but they focus on establishing a systematic method for identifying similarities, and use it to identify classes of repositories (e.g. Android versus web applications). Most of these studies use topic modeling, which is one of the approaches that we considered initially, but gave poor results in our context, but we will revisit in the future.

c. Databases of malware source code: At the time of writing this paper, there are few malware source code databases

and are rarely updated such as project *theZoo* [72]. To the best of our knowledge, there does not exist an active archive of malware source code, where malware research community can get an enough number of source code to analyze.

d. Databases of malware binaries: There are well established malware binary collection initiatives, such as Virus-total [68] which provides analysis result for a malware binary. There are also community based projects such as Virus-Bay [69] that serve as malware binary sharing platform.

e. Converting binaries to source code: A complementary approach is to try to generate the source code from the binary, but this is a very hard task. Some works [19, 20] focus on reverse engineering of the malware binary to a high-level language representation, but not source code. Some other efforts [11, 29, 59] introduce binary decompilation into readable source code. However, malware authors use sophisticated obfuscation techniques [56] [10, 73] to make it difficult to reverse engineer a binary into source code.

f. Measuring and modeling hacking activity. Some other studies analyze the underground black market of hacking activities but their starting point is security forums [18, 51, 57], and as such they study the dynamics of that community but without retrieving any malware code.

10 Conclusion

Our work capitalizes on a great missed opportunity: there are thousands of malware source code repositories on GitHub. At the same time, there is a scarcity of malware source code, which is necessary for certain research studies.

Our work is arguably the first to develop a systematic approach to extract malware source-code repositories at scale from GitHub. Our work provides two main tangible outcomes: (a) we develop SourceFinder, which identifies malware repositories with 89% precision, and (b) we create, possibly, the largest non-commercial malware source code archive with 7504 repositories. Our large scale study provide some interesting trends for both the malware repositories and the dynamics of the malware authors.

We intend to open-source both SourceFinder and the database of malware source code to maximize the impact of our work. Our ambitious vision is to become the authoritative source for malware source code for the research community by providing tools, databases, and benchmarks.

Acknowledgements

This work was supported by the UC Multicampus National Lab Collaborative Research and Training (UC NL CRT) award #LFR18548554.

References

- [1] 3vilp4wn. Hacking tool of 3vilp4wn. <https://github.com/3vilp4wn/CryptLog/>. [Online; accessed 08-February-2020].

- [2] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.
- [3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.
- [4] John Aycock. *Computer viruses and malware*, volume 22. Springer Science & Business Media, 2006.
- [5] Andrew Begel, Jan Bosch, and Margaret-Anne Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and top-coder. *IEEE Software*, 30(1):52–66, 2013.
- [6] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.
- [7] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.
- [8] Alejandro Calleja, Juan Tapiador, and Juan Caballero. A look into 30 years of malware development from a software metrics perspective. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 325–345. Springer, 2016.
- [9] Alejandro Calleja, Juan Tapiador, and Juan Caballero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 14(12):3175–3190, 2018.
- [10] Gengbiao Chen, Zhengwei Qi, Shiqiu Huang, Kangqi Ni, Yudi Zheng, Walter Binder, and Haibing Guan. A refined decompiler to generate c code with high readability. *Software: Practice and Experience*, 43(11):1337–1358, 2013.
- [11] Gengbiao Chen, Zhuo Wang, Ruoyu Zhang, Kan Zhou, Shiqiu Huang, Kangqi Ni, and Zhengwei Qi. A novel lightweight virtual machine based decompiler to generate c/c++ code with high readability. *School of Software, Shanghai Jiao Tong University, Shanghai, China*, 11, 2010.
- [12] Jingnian Chen, Houkuan Huang, Shengfeng Tian, and Youli Qu. Feature selection for text classification with naïve bayes. *Expert Systems with Applications*, 36(3):5432–5435, 2009.
- [13] Chris Stobing. ios malwares in 2014. <https://www.digitaltrends.com/computing/decrypt-2014-biggest-year-malware-yet/>. [Online; accessed 08-February-2020].
- [14] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141. IEEE, 2016.
- [15] CR4SH. Hacking tool of cr4sh. https://github.com/Cr4sh/s6_pcie_microblaze/. [Online; accessed 08-February-2020].
- [16] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286. ACM, 2012.
- [17] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.
- [18] Ashok Deb, Kristina Lerman, Emilio Ferrara, Ashok Deb, Kristina Lerman, and Emilio Ferrara. Predicting Cyber-Events by Leveraging Hacker Sentiment. *Information*, 9(11):280, nov 2018.
- [19] Lukáš Ďurfina, Jakub Křoustek, and Petr Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456. IEEE, 2013.
- [20] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. In *International Conference on Information Security and Assurance*, pages 72–86. Springer, 2011.
- [21] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.
- [22] Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Idapro for iot malware analysis? In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

- [23] Joobin Gharibshah, Evangelos E Papalexakis, and Michalis Faloutsos. Rest: A thread embedding approach for identifying and classifying user-specified information in security forums. *arXiv preprint arXiv:2001.02660*, 2020.
- [24] GitHub. Repository search for public repositories: Showing 32,107,794 available repository results. <https://github.com/search?q=is:public/>. [Online; accessed 13-October-2019].
- [25] GitHub. User search: Showing 34,149,146 available users. <https://github.com/search?q=type:user&type=Users/>. [Online; accessed 13-October-2019].
- [26] Amir Globerson, Gal Chechik, Fernando Pereira, and Naftali Tishby. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research*, 8(Oct):2265–2295, 2007.
- [27] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017.
- [28] HackerChatter. UCR hacker forum webtool for extracting useful information from security forums! <http://www.hackerchatter.org/>. [Online; accessed 22-July-2020].
- [29] Richard Healey. Source code extraction via monitoring processing of obfuscated byte code, August 27 2019. US Patent 10,394,554.
- [30] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O Hall, and Adriana Iamnitchi. Mentions of security vulnerabilities on reddit, twitter and github. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 200–207. ACM, 2019.
- [31] James Howison and Kevin Crowston. The perils and pitfalls of mining sourceforge. In *MSR*, pages 7–11. IET, 2004.
- [32] James A Jerkins. Motivating a market or regulatory solution to iot insecurity with the mirai botnet code. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–5. IEEE, 2017.
- [33] Anjali Ganesh Jivani et al. A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011.
- [34] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.
- [35] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.
- [36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.
- [37] Bence Kollanyi. Automation, algorithms, and politics: Where do bots come from? an analysis of bot codes shared on github. *International Journal of Communication*, 10:20, 2016.
- [38] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pages 957–966, 2015.
- [39] Michael J Lee, Bruce Ferwerda, Junghong Choi, Jungpil Hahn, Jae Yun Moon, and Jinwoo Kim. Github developers use rockstars to overcome overflow of news. In *CHI’13 Extended Abstracts on Human Factors in Computing Systems*, pages 133–138. ACM, 2013.
- [40] Toomas Lepik, Kaie Maennel, Margus Ernits, and Olaf Maennel. Art and automation of teaching malware reverse engineering. In *International Conference on Learning and Collaboration Technologies*, pages 461–472. Springer, 2018.
- [41] Yitan Li, Linli Xu, Fei Tian, Liang Jiang, Xiaowei Zhong, and Enhong Chen. Word embedding revisited: A new representation learning and explicit matrix factorization perspective. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- [42] Michael Frederick McTear, Zoraida Callejas, and David Griol. *The conversational interface*, volume 6. Springer, 2016.
- [43] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [44] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

- [45] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.
- [46] nlnj4sec. Pupy tool. <https://github.com/nlnj4sec/pupy/wiki/>. [Online; accessed 08-February-2020].
- [47] Nicolas Verdier. Security researcher. <https://www.linkedin.com/in/nicolas-verdier-b23950b6/>. [Online; accessed 14-February-2020].
- [48] Nikhil Gupta. Should we create a separate git repository of each project or should we keep multiple projects in a single git repo? <https://www.quora.com/>. [Online; accessed 14-February-2020].
- [49] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.
- [50] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 348–351. ACM, 2014.
- [51] Rebecca S. Portnoff, Sadia Afroz, Greg Durrett, Jonathan K. Kummerfeld, Taylor Berg-Kirkpatrick, Damon McCoy, Kirill Levchenko, and Vern Paxson. Tools for Automated Analysis of Cybercriminal Markets. *Proceedings of the 26th International Conference on World Wide Web - WWW '17*, pages 657–666, 2017.
- [52] PyGithub. A python library to use github api v3. <https://github.com/PyGithub/PyGithub/>. [Online; accessed 13-October-2019].
- [53] Austen Rainer and Stephen Gale. Evaluating the quality and quantity of data on open source software projects. In *Procs 1st int conf on open source software*, 2005.
- [54] Raj Chandel. Article on pupy. <https://www.hackingarticles.in/command-control-tool-pupy/>. [Online; accessed 08-February-2020].
- [55] Monica Rogati and Yiming Yang. High-performing feature selection for text classification. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 659–661, 2002.
- [56] Hassen Sardi, Phillip Porras, and Vinod Yegneswaran. Experiences in malware binary deobfuscation. *Virus Bulletin*, 2010.
- [57] Anna Sapienza, Sindhu Kiranmai Ernala, Alessandro Bessi, Kristina Lerman, and Emilio Ferrara. Discover: Mining online chatter for emerging cyber threats. In *Companion Proceedings of the The Web Conference 2018, WWW '18*, pages 983–990, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.
- [58] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015.
- [59] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.
- [60] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in computer virology*, 7(2):107–119, 2011.
- [61] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 314–319, 2017.
- [62] Victor RL Shen, Chin-Shan Wei, and Tony Tong-Ying Juang. Javascript malware detection using a high-level fuzzy petri net. In *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 2, pages 511–514. IEEE, 2018.
- [63] Marcus Soll and Malte Vosgerau. Classifyhub: an algorithm to classify github repositories. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 373–379. Springer, 2017.
- [64] SL Ting, WH Ip, and Albert HC Tsang. Is naive bayes a good classifier for document classification. *International Journal of Software Engineering and Its Applications*, 5(3):37–46, 2011.
- [65] Tom K. Hacking news of fahim magsi. <https://www.namepros.com/threads/hacked-by-muslim-hackers.950924/>. [Online; accessed 08-February-2020].
- [66] Tommy Hodgins. Choosing between “one project per repository” vs “multiple projects per repository” architecture. <https://hashnode.com/>. [Online; accessed 14-February-2020].
- [67] Christoph Treude, Larissa Leite, and Maurício Aniche. Unusual events in github repositories. *Journal of Systems and Software*, 142:237–247, 2018.

- [68] Virus Total. Free online virus, malware and url scanner. <https://www.virustotal.com/en>. [Online; accessed 08-February-2020].
- [69] VirusBay. A web-based, collaboration platform for malware researcher. <https://beta.virusbay.io/>. [Online; accessed 08-February-2020].
- [70] Wikipedia. Linux based botnet bashlite. <https://en.wikipedia.org/wiki/BASHLITE/>. [Online; accessed 08-February-2020].
- [71] Shuo Xu. Bayesian naïve bayes classifiers to text classification. *Journal of Information Science*, 44(1):48–59, 2018.
- [72] Y. Nativ and S. Shalev. thezoo: A live malware repository. <https://github.com/ytisf/theZoo>. [Online; accessed 08-February-2020].
- [73] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE, 2016.
- [74] Yuval Nativ. Security researcher. <https://morirt.com/>. [Online; accessed 14-February-2020].
- [75] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4):43–52, 2010.
- [76] Yu Zhang, Frank F Xu, Sha Li, Yu Meng, Xuan Wang, Qi Li, and Jiawei Han. Higitclass: Keyword-driven hierarchical classification of github repositories. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 876–885. IEEE, 2019.
- [77] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23. IEEE, 2017.
- [78] Xingsi Zhong, Yu Fu, Lu Yu, Richard Brooks, and G Kumar Venayagamoorthy. Stealthy malware traffic-not as innocent as it looks. In *2015 10th International Conference on Malicious and Unwanted Software*, pages 110–116. IEEE, 2015.

HyperLeech: Stealthy System Virtualization with Minimal Target Impact through DMA-Based Hypervisor Injection

Ralph Palutke

Friedrich-Alexander-Universität Erlangen

Matthias Wild

Friedrich-Alexander-Universität Erlangen

Simon Ruderich

Friedrich-Alexander-Universität Erlangen

Felix C. Freiling

Friedrich-Alexander-Universität Erlangen

Abstract

In the recent past, malware began to incorporate anti-forensic techniques in order to hinder analysts from gaining meaningful results. Consequently, methods that allow the stealthy analysis of a system became increasingly important.

In this paper, we present *HyperLeech*, the first approach which uses DMA to stealthily inject a thin hypervisor into the memory of a target host, transparently shifting its operation into a hardware-accelerated virtual machine. For the code injection, we make use of external PCILeech hardware to enable DMA to the target memory. Combining the advantages of hardware-supported virtualization with the benefits provided by DMA-based code injection, our approach can serve analysts as a stealthy and privileged execution layer that enables powerful live forensics and atomic memory snapshots for already running systems. Our experiments revealed that *HyperLeech* is sufficient to virtualize multi-core Linux hosts without causing significant impact on a target's processor and memory state during its installation, execution, and removal. Although our approach might be misused for malicious purposes, we conclude that it provides new knowledge to help researchers with the design of stealthy system introspection techniques that focus on preserving a target system's state.

1 Introduction

The ongoing arms race between malware authors and security practitioners lead to increasingly sophisticated approaches on both sides. Recently, malware began to incorporate anti-forensics to evade analysis. Sparks and Butler [58] presented a novel rootkit technique that subverts the memory translation process of the Windows operating system, and exploits *Translation Lookaside Buffer* (TLB) incoherencies to hide malicious memory. Palutke and Freiling [42], as well as Torrey [61], further enhanced this concept by dynamically virtualizing a victim system's view on the physical memory, relying on a kernel extension. Other approaches use *Direct Kernel Object Manipulation* (DKOM), first discussed by Butler [6],

to alter important kernel structures, as memory forensics and live analysis often rely on their integrity [5, 22, 59]. In addition, Zhang et al. [68] bypass state-of-art memory acquisition by manipulating the physical address layout on x86 platforms. Besides attacks that target software-based approaches, Rutkowska [53] demonstrated a method to attack *Direct Access Memory* (DMA)-based acquisition by remapping parts of the *Memory Mapped I/O* (MMIO) address space. Zdychowski et al. [66] listed further approaches in a recent meta study, surveying the landscape of modern anti-forensics. Approaches like these indicate the necessity for novel analysis techniques that are robust against anti-forensics.

To deliver ideal analysis results, an approach must meet two requirements which seemingly contradict each other: First, the *soundness* of a particular analysis method indicates its robustness against anti-forensics, meaning its degree of accuracy based on the actual data of the current target state. Second, a method's *target impact* implies the amount of modifications it introduces to a target's memory and processor state during its installation, operation, and removal. From a forensics point of view, a low target impact is desirable, as it prevents both a potential loss of evidence and the chance for evasive malware to alter its behavior [31]. Running an analysis tool at the same or even a lower privileged domain gives malware the chance to intercept its functionality and falsify results. Consequently, a sound analysis cannot be guaranteed. To keep control over a system's operation, security software steadily migrated to higher privileged layers [32]. In contrast to malware infections, the deployment of privileged analysis software mostly depends on a system's regular loading mechanisms. These have a quite significant impact on the target state and usually require root access, both disadvantageous from a forensics perspective. Furthermore, analysis methods are usually deployed after a system has been infected, which gives malware the chance to tamper with their installation. Hence, analysts began to use increasingly stealthy approaches to conceal the deployment of their methods. Stüttgen and Cohen [60] inject a minimal memory acquisition module into an already existing host kernel module with only a small target impact.

Besides the installation of an analysis method, both its execution and removal, as well as the extraction of results, which often makes use of existing communication channels, alter the target state to an even higher degree. In addition, these communication channels might already be compromised, so that the integrity of the transferred data cannot be guaranteed.

With the rise of anti-forensics, security practitioners started to use DMA from external hardware in order to analyze a system [7, 15, 36, 44]. This allows the transparent access of a system's memory without notably impacting its state, as DMA does not interfere with a processor's operation. Since these devices are often hot pluggable, DMA-based approaches offer a significant advantage when targeting production systems, where down times are often not acceptable. As hot plugging allows a method to be deployed even after the infection of a system, it is especially useful for malware analysis. In addition, DMA usually bypasses authorization checks enforced by the operating system. As a downside, Gruhn and Freiling [21] showed that these approaches suffer from a lack of atomicity, since the target is not suspended during the analysis or acquisition process. Consequently, they cannot produce fully sound analysis results.

Virtualization-based approaches provide the transparent analysis of a system from the more privileged hypervisor layer. The respective target is booted inside a virtualized execution environment (respectively VM), enabling the isolated analysis of the system through *Virtual Machine Introspection* (VMI) [18]. Since investigators are mostly confronted with already infected systems running on bare metal, these cannot be virtualized by conventional technologies like KVM [20] or Xen [4], however. This led analysts to use *on-the-fly virtualization*, initially introduced by Rutkowska [52] and Zovi [69], which installs a thin hypervisor through a kernel driver, and migrates the running system into a hardware-accelerated VM for further analysis [29, 39, 47, 65]. Although on-the-fly virtualization greatly improves the analysis of a system, it falls short in several categories. Loading a kernel driver requires root privileges and has significant impact on the target state. Furthermore, an already infected kernel might subvert the installation process altogether.

In this paper, we present *HyperLeech*, the first approach combining transparent DMA-based code injection and on-the-fly virtualization. In contrast to existing solutions, our approach enables the sound analysis of a target system with negligible impact on its processor and memory state. In detail, we

- are the first to use DMA from an external PCILeech device to stealthily inject a hypervisor into a target's memory, bypassing common access restrictions,
- use Intel's *Virtual Machine Extensions* (VMXs) to virtualize a running target by transparently shifting it into a hardware-accelerated VM, and hide our system by set-

ting up *Extended Page Tables* (EPTs), providing an abstraction of the physical memory,

- devise the process of removing our system without leaving detectable traces,
- implement a prototype that is capable of virtualizing running multi-core Linux hosts without notably impacting the target's processor and memory state,
- evaluate the target impact caused by the injection, execution and removal of our system,
- point out the performance impact caused by the injection of our system, and
- discuss possible mitigation strategies, as our approach might be misused as a powerful toolkit.

The remainder of this paper is outlined as follows: Section 2 provides fundamental background knowledge that is necessary to understand our design concepts. In Section 3, we present an architectural overview of the HyperLeech system, and describe its injection and removal. Section 4 evaluates the impact on both the target's state and performance, and discusses possible mitigation strategies. Section 5 briefly surveys related work and possible use cases. Concluding remarks and future research directions are given in Section 6.

2 Technical Background

For a better understanding of our design choices, we briefly outline important technical fundamentals. Consequently, we introduce the PCILeech framework (Section 2.1), explain the mechanics of hardware-supported virtualization provided by Intel's VT-x (Section 2.2), and shed light on the *Advanced Programmable Interrupt Controller* (APIC) (Section 2.3). Readers familiar with the topics can skip these sections.

2.1 PCILeech

Originally developed by Frisk [15], the PCILeech project is a generic attack framework that allows external devices to use DMA over *Peripheral Component Interconnect Express* (PCIe) to inject code into the physical memory of a target system. Due to PCIe offering hot plug functionality, a variety of PCILeech devices can be attached to a system at runtime. Similarly, such devices can be unplugged at any time without causing significant interruptions. PCILeech supports various hardware configurations which need to be flashed with dedicated firmware. For this work, we made use of the *PCIe Screamer* device [3] which is based on the XC7A35T Xilinx 7 *Field-Programmable Gate Array* (FPGA) providing native 64-bit DMA with access rates about 100 MB/s. Over the *Universal Serial Bus 3* (USB3) interface, the device is connected to an external controller system which is used to

control the PCILeech software. To attach PCIe Screamer to a free PCIe slot of the target host, some systems might require specific adapters due to different form factors of PCIe (e.g., Express Card, mPCIe, Thunderbolt). For HyperLeech, we only made use of PCILeech’s native DMA support to inject our custom hypervisor into a running target system. Hence, no additional software needs to be deployed on the target side.

2.2 Intel VT-x

To improve the performance of VMs, modern processors provide hardware-supported virtualization. Intel [24, vol. 3C] introduced several *Virtual Machine Extensions* (VMXs) which expand a processor’s instruction set to allow unmodified guests to be executed inside a hardware-accelerated VM. It provides the new processor mode *VMX operation* which is further divided into the execution modes *VMX root* and *VMX non-root*. The former describes a privileged mode that runs a hypervisor (or *Virtual Machine Monitor* (VMM)), used to control the VMs and schedule hardware resources. VMX non-root, on the other hand, serves unmodified *guest* systems as a transparent and restricted execution environment. The processor uses *VMX transitions* to switch between the two operation modes. With the occurrence of certain events (e.g., accesses to specific registers, execution of restricted instructions, or the interaction with emulated devices) in VMX non-root mode, the processor generates a *VM exit* which transfers control to the hypervisor. Subsequently, the hypervisor has the chance to handle the fault and resume the guest. To launch and control a VM, the hypervisor must configure a *Virtual Machine Control Structure* (VMCS) for each core. Besides comprising the entire state of the guest, this central management structure determines the events that are to be intercepted by the hypervisor.

Next to VMX, Intel processors provide *Extended Page Tables* (EPTs) that support the virtualization of a VM’s physical memory. When enabled, a second level address translation maps the guest’s physical memory to the real memory of the host machine. Similar to the conventional paging structures, EPTs provide several access flags that prohibit unauthorized memory accesses. Breaching these access privileges leads to an *EPT violation* which is intercepted and handled by the hypervisor. This gives the hypervisor the chance to restrict the guest from accessing certain memory regions.

2.3 Intel APIC

With the emergence of *Symmetric Multiprocessing* (SMP) architectures, Intel introduced the APIC system to deliver and control external interrupts. Its architecture consists of two components which communicate over the system bus. The *I/O APIC* routes external interrupts to one or more *Local Advanced Programmable Interrupt Controllers* (LAPICs), each belonging to a particular processor core. The LAPICs

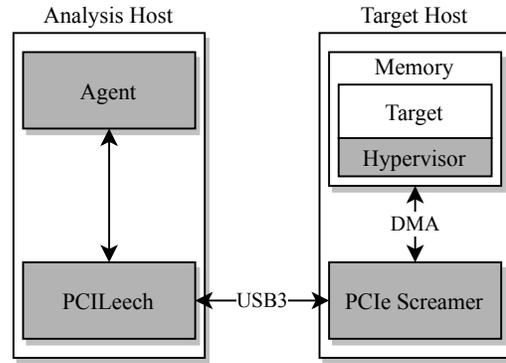


Figure 1: Architectural overview of the HyperLeech system.

receive interrupts not only from the I/O APIC, but also from the processors’ interrupt pins and other internal sources, and forward them to their respective cores for specific handling. The LAPIC appears as a memory-mapped device, providing its physical base address through the `IA32_APIC_BASE` *Model Specific Register* (MSR). The kernel initializes this register by parsing the host’s *Advanced Configuration and Power Interface* (ACPI) tables during the early boot phase. Over time, Intel introduced several successors which enhanced the design of the APIC. While the xAPIC only brought a few minor changes, the x2APIC appears as the latest iteration which is accessed through certain MSRs instead of MMIO. Both modes are supported by modern processors, and can be switched by specifying a certain bit in the `IA32_APIC_BASE` MSR. This requires the system to be rebooted, however.

Besides interrupts triggered by external devices, the LAPIC provides the possibility to generate *Non-maskable Interrupts* (NMIs). In contrast to maskable interrupts, NMI delivery cannot be trivially deactivated. Each LAPIC provides several *Local Vector Table* (LVT) registers that are used to configure the delivery of different NMI types. Among others, these include timer interrupts, thermal sensor interrupts, and performance counter overflows. The *mask bit* in an LVT allows to disable the delivery of the corresponding NMI type by preventing the LAPIC from forwarding the interrupt to its processor. Once set, further incoming NMIs of the same type set the *pending bit* in the same LVT to signal an outstanding interrupt. The pending NMI is not delivered to the processor until the mask bit in the respective LVT has been cleared. While being disabled only one upcoming NMI can be kept pending. Any additional NMI is lost.

3 System Overview

This section provides an architectural overview of the HyperLeech system, and illustrates its injection (Section 3.2) and removal (Section 3.3) mechanisms. The basic architecture

of our prototype comprises several components which are grayly depicted in Figure 1. HyperLeech targets a physical host which is subject to be analyzed. As the target will exclusively be accessed from external hardware, no login credentials are required to inject our *hypervisor* into its memory. To access the memory of the target host, we flash the PCILeech firmware to the *PCIe Screamer* FPGA, and attach it to a free PCIe slot on the target side. PCIe Screamer allows native 64-bit DMA operations, and thus access to the target's entire physical memory. Over USB3, PCIe Screamer is connected to an analysis machine that is used to execute the controlling *agent* software. The agent, written in Go, serves the analyst as an interface for controlling our hypervisor through the *PCILeech* host software.

3.1 Mode of Operation

For the installation and removal of our hypervisor, the agent uses DMA to inject multiple code stages into the target memory. This is possible as x86-64 ensures cache coherency regarding DMA operations, preventing processors from retrieving inconsistent data [24, chap. 11.3.2]. The stages are designed to preserve memory and processor state, so that the target is not notably impacted by the injection (see Section 4.1). Apart from this, we outsource most computational tasks to the remote agent in order to further reduce target modifications.

Prior to the code injection, the agent needs to determine the location of the target kernel. Due to the usage of *Kernel Address Space Randomization* (KASLR), modern Linux kernels are randomly placed in physical memory. Therefore, the agent scans the entire physical memory until it matches a pre-registered signature of the kernel's first code page. To correctly terminate the scanning process, the agent issues DMA read operations to probe the amount of memory installed in the target host. As certain memory ranges do not respond to DMA (e.g., MMIO areas which are used to map certain devices), a timeout is employed on each read operation to prevent the agent from stalling.

Once the kernel is found, the agent writes special *injection stages* (see Section 3.2) into the target's physical memory. These stages hijack the control flow of the target kernel, and subsequently install a thin hypervisor that virtualizes the system at runtime. Despite being able to write arbitrary memory over DMA, taking over the system's operation is mandatory to actually execute the injected code. For a stealthy operation, we designed a lightweight, VMX-based custom hypervisor that mostly avoids any interference with the target's operation. Limited to DMA, the agent cannot obtain contextual information of the running system, as it is restricted to a physical view. Therefore, the agent risks potentially corrupting the target by overwriting the memory which is currently in use. To minimize this risk, the agent searches for regions that are unlikely to be used. During our research, we determined the first two KiBs of a Linux kernel's code segment to be the perfect injection

spot as it mostly consists of `nop` instructions which do not have any functional purpose. As of that, modifying this *nop area* does not corrupt the kernel's execution. Prior to overwriting any memory, the agent stores the original content to the analysis machine, so it can be restored in due time. Since the `nop` area does not hold enough space for the actual hypervisor, the stages request the target kernel to allocate further memory. This is not an issue from a forensics perspective, as occupying currently unused memory should not corrupt evidential data in most cases. After receiving the necessary information, the agent sets up the hypervisor's memory layout, and copies the appropriate page tables and the binary of the hypervisor to the previously allocated memory. Withdrawing control from the target leads the hypervisor to take over and virtualize the running cores. At this point, the hypervisor is in full control, ready to fulfill stealthy analysis tasks or perform memory forensics. Eventually, the agent removes the injection stages, restoring the original target memory.

Using DMA, an analyst can now instruct the agent to transparently interact with the hypervisor, allowing to send commands or receive data. Compared to conventional communication channels that rely on the file system or a network card, exchanging data over DMA is both stealthier and less intrusive to the target state. Especially for targets that might be compromised, covert communication is an important requirement for the integrity of the transferred data.

To deinstall our system, the agent overwrites parts of the target's kernel space with *removal stages* (see Section 3.3) which have the purpose of transparently devirtualizing the system. After that, the hypervisor signals the agent to restore the overwritten memory including the hypervisor area. At this point, the target is resumed without leaving any traces of our system.

3.2 Injection

We designed the injection process in multiple *injection stages*, so that no relevant data is corrupted during the virtualization of the target system. We minimized dependencies on the target kernel, as the system could have already been compromised, and thus subvert the injection process. One exception is to query the target kernel to allocate some memory for the *hypervisor area*. This seems acceptable, however, as we doubt that malware could forge the allocation without risking severe system failures. Figure 2 gives an overview of the injection. While stage I1 only consists of a few bytes overwriting a kernel function to hijack the control flow, I2 and I3 are written to the previously mentioned `nop` area. In contrast, I4 and I5 are directly placed within the hypervisor area. Upcoming, we shed light on the individual injection stages.

Stage I1: Control Flow Hijacking Once the target kernel's base address was found, an injection location which allows to take over a processor's execution is chosen. In contrast

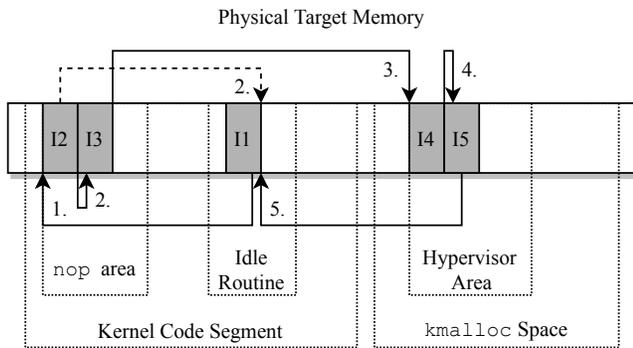


Figure 2: View of physical memory after the injection stages I1 to I5 were written to the target memory. The hypervisor area was vicariously allocated by the target kernel. Arrows represent jumps to subsequent stages. The dashed arrow symbolizes the abort of the injection, jumping back to the kernel’s idle routine.

to the base of the kernel which differs after each boot due to KASLR, offsets within its code segment only depend on the actual kernel version. This is because the Linux kernel is linearly mapped in both physical and virtual memory. Consequently, addresses of arbitrary kernel symbols can be statically computed for a particular target by adding the respective offsets to the previously determined kernel base address.

To actually hijack the target kernel, we chose to hook its idle routine (`intel_idle`), as it is regularly executed by each processor and allows to run code with ring 0 privileges which are required for the virtualization. Other valid injection spots include a system’s scheduler as well as its interrupt handlers. For the later, special care has to be taken since the injected code needs to run in interrupt context, however. To decrease the latency a processor takes to enter our hook, even multiple injection locations could be selected. As long as the hook is placed within the target’s kernel space and is regularly executed, almost any location could be used to hijack the control flow. This prevents the target system from mitigating our injection by monitoring specific memory regions.

As soon as a core starts idling, `intel_idle` is invoked, entering a sleeping state to save power until it is woken up again. Before placing the hook, the agent saves the respective memory to be able to restore it later on. Acquiring memory over non-atomic protocols like DMA introduces a race condition, as the kernel might alter the area before it enters the hook. As the kernel’s code segment should be mapped non-writable at all times, it usually is never modified, however. Nevertheless, a custom kernel might remap its code ranges to be writable. Even mainline kernels might sometimes alter parts of their code segments when using instrumentation frameworks like `Ftrace`. When activated, `Ftrace` dynamically modifies the first few bytes of a kernel function, so that subsequent invocations

detour to a custom handler before returning to the originally intended code. Therefore, we avoided to overwrite these first few bytes with our hook. Instead, the agent replaces the following bytes with a relative jump to stage I2. As we could have chosen any other spot in the kernel code, monitoring the idle loop is not a reliable method to generically detect our approach. To virtualize every core, the hook must remain until all processors have passed the residual stages. Our current implementation relies on target kernel functionality to determine the number of target cores (`num_online_cpus`). However, enumerating the *Non-Uniform Memory Access* (NUMA) hierarchy could possibly provide a more target independent way. After entering the hook, the core jumps to stage I2.

Stage I2: Processor Serialization Until having allocated further memory for the hypervisor, we use the kernel’s stack to temporarily store register content that is about to be modified, preventing a loss of processor state. As discussed in Section 4.1, this appears to be insignificant from a forensics perspective. The subsequent process is sequentialized by the possession of a global lock which prevents the target cores from concurrently entering the following stages. In case the lock has already been occupied, a core immediately detours to a specifically prepared trampoline that is responsible for executing the instructions overwritten by our hook, before resuming the original execution within the idle loop (dashed arrow in Figure 2). Thus, no processor stalls while the lock is held by another core. The trampoline mechanism is also used for processors that were already virtualized, as these must be prevented from reentering the subsequent stages. To determine the current processor’s virtualization state, I2 attempts to force a VM exit which is only generated in VMX non-root mode. In case of an already virtualized core, this forces a context switch to the hypervisor which in turn informs I2 about the current processor’s operation mode. Whenever a non-virtualized core acquires the lock, it is allowed to enter the subsequent stages, eventually leading to its virtualization.

Stage I3: Hypervisor Setup To prevent the target from disrupting the installation of our hypervisor, I3 temporarily disables all interrupts until the processor reaches the end of I4. Besides maskable interrupts, modern processors support NMIs for critical asynchronous event delivery like hardware interrupts or watchdogs. These special kind of interrupts cannot be trivially disabled with the `cli` instruction, however. To temporarily deactivate NMI delivery, I3 reconfigures each processor’s LAPIC, disabling the valid flags of its LVT registers (see Section 2.3). Additionally, it consults the `IA32_APIC_BASE` MSR to determine the system’s current APIC mode, as both xAPIC and x2APIC are supported by modern processors. Consequently, our implementation offers different ways to access a LAPIC. As NMIs must be disabled before switching to the hypervisor’s memory layout, there

is no way to map the LAPICs' physical addresses. Therefore, we decided to rely on the kernel's `APIC_BASE` symbol to make use of the already established kernel mapping. From a forensics view, we do not expect this symbol to be a critical target dependency, as it is defined as a kernel constant.

For the memory of the actual hypervisor, I3 manually calls the kernel's `kmalloc` function to allocate additional space. Our experiments revealed that a single two MiB page seems to be a sufficient size. Most of the hypervisor area serves for dynamic memory allocations during the setup of the hypervisor. Other parts are used to store its code, data, stacks, and page tables. The resulting base address of the hypervisor area is then translated to its physical counterpart, and provided to the agent using DMA. After receiving the information, the agent copies relevant parts of the hypervisor to the newly allocated area. These include a custom memory layout which exclusively maps the hypervisor and the injection stages. Depending on the specific use case, mapping certain parts of the guest's address space might be conceivable. Eventually, the processor's TLBs are flushed, the newly created memory layout is enabled, and *Process Context Identifiers* (PCIDs) are deactivated to prevent caching leftovers.

Stage I4: NMI Handling Entering stage I4 first sets up a processor individual stack which is subsequently used to store the state of the core. To prevent stack overflows from corrupting any memory, the stacks are surrounded by non-presently mapped pages.

Before NMIs can be safely reenables, I4 registers a custom handler which ensures NMIs that would otherwise be lost to be reinjected into the guest. This additionally requires the installation of both a custom *Global Descriptor Table* (GDT) and *Interrupt Descriptor Table* (IDT). During the execution of the hypervisor, the handler records upcoming NMIs within a bitmap which indicates if a processor has an NMI pending. To distinguish the running cores, and thus mark the correct bit within the pending bitmap, the hypervisor's NMI handler compares the stack pointer of the current processor with the hypervisor's stack ranges. As every virtualized core has its own stack which is known to the hypervisor, these information can be used to distinguish the processors without the need to consult the target kernel. Until properly acknowledged by clearing the pending flag in the respective LVT, an NMI is not forwarded to the corresponding processor core. This, however, prevents further NMIs of the same type from being delivered independent of the valid flag of the corresponding LVT register. As a solution, the hypervisor consults the pending bitmap to determine if an NMI must be reinjected to the current virtualized core every time it is about to reenter the guest. This is necessary, as watchdog tasks could potentially misbehave due to missing NMIs, and thus corrupt the target state. To inject an NMI into the guest, the hypervisor sets the *valid flag* of the *VM-entry interruption-information field* within the corresponding VMCS, and specifies the *interruption type*

field to indicate an NMI. This automatically invokes the guest kernel's NMI handler as soon as the guest is resumed. From the perspective of the guest, it is not distinguishable if an NMI appeared during its own operation or due to an injection from our hypervisor. The guest's NMI handler then clears the pending flag in the respective LVT, allowing NMIs of the same type to be delivered again. Lastly, I4 reenables all interrupts, and jumps to the final stage.

Stage I5: Processor Virtualization To preserve the full register state of a processor, all previous stages had to be implemented in plain assembly code. With a custom stack being set up during the previous stage, I5 is implemented in the high-level programming language C. The stage is responsible for the setup of our hypervisor and the actual virtualization of a processor. Consequently, it allocates relevant data structures like the VMCS within the hypervisor area. With all the preparations done, the processor releases the global lock before entering the virtualization process. Provided *Intel Virtualization Technology* (VT-x) was not deactivated in the *Unified Extensible Firmware Interface* (UEFI) or BIOS settings (which is not the case in most modern machines), I5 enables VMX operation, and copies the previously saved processor state to the VMCS. Similar to a hypervisor rootkit [42, 52, 69], this allows to transparently launch the target system inside a hardware-assisted VM, and resume its original execution with its current state. To remain stealthy, we configure the VMCS so that the hypervisor intercepts only a minimal set of events. Thus, hardware is directly passed to the guest without the hypervisor's interference. This increases the overall performance of the guest while reducing detectable side channels. Depending on the specific use case, it might be necessary to configure the interception of additional events, however. Due to the hypervisor area being allocated by the target kernel, it should never be accessed by accident. To prevent the target from purposely accessing the hypervisor area, we set up EPTs to redirect read accesses to a 4 KiB *guard page*. Write accesses are configured to generate an EPT violation in order to keep the originally stored memory on the analysis machine up to date. Consequently, the hypervisor intercepts the write accesses, and notifies the agent to update its copy. The remaining physical address space is identity mapped. Eventually, the hypervisor resumes the target's original execution inside the idle routine, now running as a virtualized guest. As we chose a symmetric hypervisor design, the entire injection process needs to be repeated for each core. From there on, the hypervisor can be used for any specific task while stealthily controlling the target system.

3.3 Removal

Similar to the injection, the removal of the hypervisor requires the agent to copy several *removal stages* to the target memory. These stages devirtualize the target, and clean up its memory

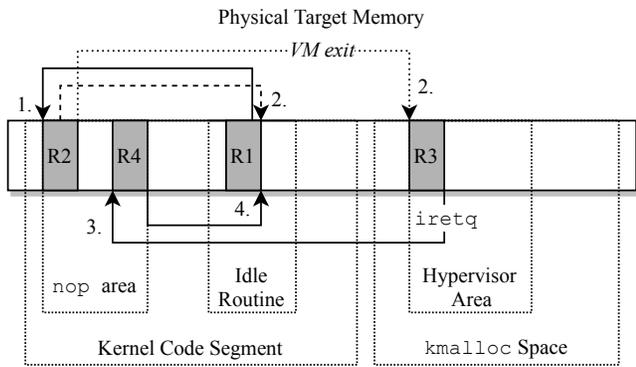


Figure 3: View of physical memory after the removal stages R1 to R4 were written to the target memory. The arrows represent jumps to subsequent stages. The dashed arrow symbolizes the abort of the removal process, returning to the kernel’s idle routine. The dotted arrow depicts a mode switch to the hypervisor running in VMX root mode.

and processor state. Rebooting the system will also remove the hypervisor, as it only resides in memory and is currently not configured to intercept and emulate system shutdowns. This is not a conceptual limitation, however. Figure 3 illustrates the injected removal stages within the physical memory of the target host. Once again, the first stage R1 appears as a hook within a permanently invoked kernel function, enabling HyperLeech to take over control. R2 and R4 are placed within the nop area, and R3 is part of the hypervisor’s code base. The remains of this section provide brief information about the individual removal stages.

Stage R1: Control Flow Hijacking To devirtualize the target system, a context switch to VMX root mode is mandatory. As we cannot rely on the guest to trigger a VM exit, we decided to reinstall a hook within the idle routine (`intel_idle`) to withdraw control from the target kernel. Like with the injection process, entering the hook transfers each processor to the subsequent stages. Once again, care had to be taken to inject the following stages before installing the hook.

Stage R2: Processor Serialization Similarly to the injection process, the trampoline mechanism prevents certain cores from entering the subsequent stages (dashed arrow in Figure 3). This time, however, already devirtualized processors detour to our trampoline, execute the overwritten instructions, and return to the idle loop. In addition, a global lock once again ensures the serialization of the following stages. Failing to acquire the global lock also results in the immediate return to the idle routine after detouring to the trampoline. Here on after, R2 forces another context switch to the hypervisor which provides the next stage (dotted arrow in Figure 3).

Stage R3: Processor Devirtualization Stage R3, now being executed in VMX root mode, first determines if the guest’s current instruction pointer refers to the code of R2. This verifies that the context switch was indeed caused by R2 indicating a valid unload process. Although standard kernels would not map any legitimate code to their nop area, a custom kernel might deviate from this behavior, forcing our hypervisor to be unloaded. Therefore, the agent is required to approve the removal process beforehand. Otherwise, the hypervisor simply ignores the unload request, even if it came from the correct address range. In case of a legitimate removal, stage R3 disables interrupt delivery. NMIs that occurred up to this point would be lost after the hypervisor’s deinstallation, as these could not be reinjected into the guest anymore. This, however, would block further NMIs from being delivered, as the target kernel would not be aware of the pending interrupt. To avoid this issue, our hypervisor aborts the devirtualization in case of a pending NMI, reinjects it, and waits for the target to reenter the idle loop, restarting the removal process. As this time window is relatively small, NMIs barely came across during our experiments, however. After disabling interrupts, VMX operation can safely be terminated, effectively devirtualizing the core. Then, R3 restores the suspended guest processor state by consulting the respective VMCS. Via the `iretq` instruction, the remaining processor state is restored and control detours to stage R4.

Stage R4: Hypervisor Cleanup Once a processor was devirtualized, stage R4 restores the memory layout of the target and releases the global lock, allowing further cores to proceed with the removal. Afterwards, interrupts are reenabled, as these can now be handled by the target. To conceal any traces, the last core entering R4 signals the agent to restore the hypervisor area. As the saved copy was constantly updated during the interception of write accesses, it always contains the current state. It is then freed by issuing the kernel’s `kfree` function. From there on, the target has no chance to detect any evidence of the previously existing hypervisor. Subsequently, R4 jumps back into the idle routine, resuming the original execution of the target. After all cores have been devirtualized, the agent restores the memory of the injected removal stages, preventing traces from being left over.

4 Discussion

We tested our implementation on a target host running an Intel Sandy Bridge i5-2400 processor supporting VT-x, with four GiB of memory, and a Fujitsu D3062-A1 motherboard. We installed Debian Stretch with the Linux kernel version 4.9.88-1.deb9u1 as its operating system. Since HyperLeech is mostly operating system agnostic, only minor adaptations are required to support newer target kernels. Inserting the PCIe Screamer card [3] which is flashed with the PCILeech firmware version

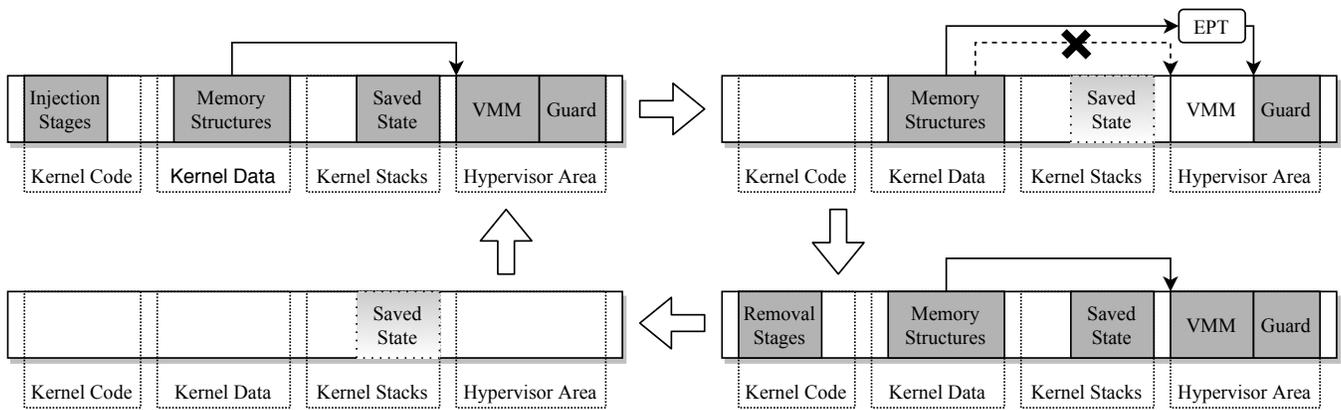


Figure 4: Impact on the target memory during the injection (top left), execution (top right), and removal (bottom right) of the HyperLeech system. Bottom left depicts the memory state after our system was fully removed. *Memory structures* represent kernel structures that reference the hypervisor area due to the allocation via `kmalloc`. During the execution, these references are redirected to the hypervisor’s *guard page*. *Saved state* depicts processor registers that are temporarily saved on the kernel stacks.

3.5 [17], grants DMA to the target’s memory. On the analysis machine, we used Windows 10 Enterprise (revision 1709) which required the installation of the FTDI USB drivers [1] to communicate with the PCILeech firmware.

4.1 Target Impact

This section discusses the modifications of the target’s processor and memory state, which arise due to the injection, execution, and removal of the HyperLeech system. Our main design consideration was to reduce the impact on the target while minimizing dependencies on kernel functionality.

Injection During the installation and removal of our system, the agent replaces certain parts of the target memory with the *injection stages* (Figure 4, top left). To preserve the original memory, the agent saves the respective areas for later restoration. Executing injected code within the target kernel typically leads to further modifications. Thus, each stage is designed to best preserve the target’s memory and processor state. As multiple cores could enter the first two injection stages in parallel, we use the kernel stacks to store processor state (called *saved state* in Figure 4) that is about to be modified. This also simplifies our implementation, as it renders error-prone synchronization mechanisms obsolete. Pushing state onto the kernel stacks cannot corrupt data that is still required by the target, as even red-zones are disabled for proper interrupt handling. Upon entering stage I3, the remaining injection is serialized by a global lock. Consequently, we store further data directly within the memory of stage I3 (and thus in the kernel’s `nop` page), as it is restored anyway and does not require any synchronization. With I4 being entered, custom stacks for each processor are allocated within the hypervisor area. These are subsequently used to store a processor’s state.

Execution After HyperLeech has been installed (Figure 4, top right), the agent restores the injection stages with their original content. This leaves only minor modifications of the *kernel stacks* and a few *memory structures* which reference the hypervisor area due to the allocation via `kmalloc`. As the configuration of EPTs redirects all read accesses targeting the hypervisor area to a *guard page*, following these references won’t find any suspicious traces. The target kernel assumes this area to be legitimately in use anyway, and usually won’t ever access it until it is freed again. Write operations that target the hypervisor area are intercepted, and the originally stored copy is updated on the analysis machine. We consider the modifications of the kernel stacks to be practically undetectable, as these are almost instantly overwritten by regular kernel usage once the target is resumed.

Removal Removing our hypervisor requires the injection of the *removal stages* which are repeatedly used to devirtualize the system (Figure 4, bottom right). As with the injection, R1 hijacks the target’s control flow, while R2 serializes the remaining process. Once again, the kernel stacks are used to preserve target processor state that is about to be modified in the meantime. R3 devirtualizes a core and restores the current guest processor state. The final core entering stage R4 frees the hypervisor memory after its has been restored by the agent. As the stored copy was constantly updated during the hypervisor’s operation, it always contains the current content. Here on after, the hypervisor area is queued back into the kernel’s slab allocator, effectively deleting the referencing *memory structures*. As a last step, the *removal stages* are overwritten by the agent, restoring the original target memory (Figure 4, bottom left). With the exception of the negligible modifications of the kernel stacks, no processor or memory state is ever

lost during the injection, execution, or removal of our system. Moreover, the overwritten parts of the kernel stacks do not contain any relevant data, and the modifications are almost instantly overwritten as soon as the target resumes its execution. Especially for the purpose of memory forensics, where evidential integrity plays an important factor, HyperLeech seems to be a promising step in the right direction.

PCIe Screamer Although the injection, operation, and removal seemingly have no notable target impact, attaching the PCIe Screamer card introduces detectable modifications. This is because the PCIe bus is enumerated whenever the kernel registers a new device. The enumeration introduces changes to the file system and leads to modifications of the memory and processor state. For the target, however, PCIe Screamer only appears as a Xilinx ethernet adapter which could further be adapted by altering its device id. Thus, the target cannot refer the device to our system, as it is not distinguishable from any legitimately added hardware. Section 6 presents an alternative injection method that avoids the necessity of attaching hardware altogether, so that even changes caused by the enumeration could be prevented.

4.2 Performance Impact

This section summarizes the performance impact of the injection. In this course, we measure the duration of the virtualization of each target core (*Core X*), and compare the cumulative *sum* to the time required by the *full* injection process. For better results, we repeated the measurements for five iterations after resetting the target each time. Table 1 summarizes our results. Comparing the measurements of the individual processors, the first core takes three times longer to finish the injection. This is because the first core is responsible for executing additional tasks, e.g., requesting the target to allocate the hypervisor area, waiting for the agent to establish a custom memory layout, and copying the hypervisor to the target memory. Compared to the cumulative *sum* of the measured durations, finishing the *full* target virtualization lasted significantly longer. This is due to preparation and cleanup steps of the agent, as well as the virtualization being serialized by a global lock during the second stage (see Section 3.2). Consequently, only one core at a time is able to progress through the remaining stages. While the lock is occupied, all other processors resume the target’s original execution until they retry the virtualization when entering the idle routine the next time. Although the serial approach leads the entire injection to last longer, no processor has to stall its original work.

As we designed our hypervisor to intercept only a minimal set of events, its performance impact during its execution appears to be minimal [2]. Depending on the actual use case, this overhead might change, however.

Table 1: The time each *core* takes to be virtualized, measured over five different runs. While *Sum* cumulates the durations of each processor run, *Full* informs about the total duration of the entire target virtualization. The bottom row visualizes the mean values of the individual runs.

Core 0	Core 1	Core 2	Core 3	Sum	Full
225 ms	66 ms	64 ms	71 ms	426 ms	6784 ms
233 ms	66 ms	63 ms	71 ms	433 ms	1652 ms
545 ms	76 ms	76 ms	65 ms	762 ms	1561 ms
597 ms	72 ms	87 ms	64 ms	821 ms	4029 ms
249 ms	71 ms	65 ms	64 ms	449 ms	1627 ms
369 ms	70 ms	71 ms	67 ms	578 ms	3132 ms

4.3 Memory Acquisition

To counter anti-forensics, analysts often acquire a system’s volatile memory for subsequent analysis [32]. This has the advantage that malware cannot actively hide once the snapshot has been acquired. Vömel and Freiling [62] introduced the three criteria *correctness*, *atomicity*, and *integrity* to assess the quality of an acquisition method. *Correctness* determines the differences between the dump and the actual memory content acquired at a certain time. Thus, malware that tampers with the acquisition process could impair the correctness of a dump [5, 22, 68]. To consider a memory dump as *atomic*, the acquisition process must not be affected by the target system’s concurrent activity. Since memory is mostly acquired at a system’s runtime, a correct memory image does not inevitably imply atomicity. Lastly, the criterion *integrity* measures to what extent memory content is altered by the acquisition method itself. As most acquisition software directly runs on the target host, certain memory needs to be overwritten by its own code and data. These criteria can be mapped to the analysis requirements *soundness* and *target impact*, defined in Section 1. While a sound analysis method must inherently be correct and atomic, the target impact can be seen as the pendant to the criterion integrity.

Various approaches emphasized the benefits of acquiring volatile memory through on-the-fly virtualization [29, 39, 65]. Most rely on a copy-on-write mechanism that provides both full atomicity and correctness for the acquired image. However, full integrity cannot be achieved, as the installation of these systems has a substantial impact on the particular target state. Other approaches perform DMA to acquire memory without having a significant impact on the target state [7, 36]. However, as DMA does not interfere with a processor’s execution, atomicity cannot be guaranteed.

HyperLeech was specifically designed to minimize the target impact, and thus advances memory acquisition to achieve much better integrity compared to existing approaches. The previously mentioned copy-on-write mechanism could be integrated into our system. While it would also achieve full atomicity and correctness, deploying our system to the target

host would require severely less state modifications (see Section 4.1). Section 6 discusses further enhancements that might allow memory acquisition to even fully satisfy all three criteria. Further conceivable use cases are outlined in Section 5.

4.4 Mitigation

As our system could also be misused as a malicious hypervisor rootkit [42, 52, 69], we discuss approaches that either prevent or at least detect its presence.

Prevention As HyperLeech relies on hardware support to set up a VM, the virtualization of the target cores can be mitigated by disabling VT-x in the UEFI or BIOS respectively. This, however, would also deprive the possibility to run legitimate VMs, and is thus often not a valid option. Alternatively, a target system might entirely prohibit the attachment of new PCIe devices, preventing our agent from accessing the target memory. However, this would also hinder a user from legitimately connecting additional PCIe hardware.

A better solution might be the restriction of DMA from untrusted sources. Modern systems provide an *Input/Output Memory Management Unit* (IOMMU) which functions similar to a standard *Memory Management Unit* (MMU), as it traverses certain mapping tables which specify memory access permissions. In contrast to regular memory accesses, an IOMMU controls DMA issued by devices, however. Hence, the target could configure the IOMMU to protect its kernel memory from illegitimate accesses. Despite an IOMMU being the best way to prevent DMA-based injections, most modern systems still do not enable it by default. This includes not only recent Linux distributions, but also Microsoft's latest operating system Windows 10 [8]. Only Apple enforces the activation and usage of an IOMMU since MacOS High Sierra [16]. However, even in case an IOMMU is in use, it has to be properly configured by the kernel and firmware respectively. Recently, Marketos et al. [37] elaborated how PCIe features like *Address Translation Services* (ATS) might enable a malicious device to act benignly, effectively bypassing an even properly configured IOMMU. According to Morgan et al. [40], a large part of today's machines grants DMA to all peripherals prior to the configuration of the IOMMU during the boot sequence for compatibility reasons. Furthermore, Markuze et al. [38] discuss the possibility to exploit trusted devices, as these are often not restricted by the IOMMU. These publications show that even the usage of a correctly configured IOMMU might not be able to prevent our approach.

As a target's processor, chipset, and mainboard must support the IOMMU in the first place, this might not even be a possibility for older machines. Instead, a system might clear the *bus master enable* flag to disable DMA for specific devices or upstream bridges. Nevertheless, Windows 10 seems to be the only system which, by default, uses this mechanism

to prevent hot-pluggable devices from using DMA during screen locks, as stated by Delaunay [8].

Detection Since the entire installation only requires a few seconds (see Section 4.2), and no significant target state is altered, detecting our system during the injection phase seems unlikely. As soon as the target has been fully virtualized, meaningful changes made to the kernel are reverted (see Section 4.1). Here on after, detecting HyperLeech is practically limited to finding indications of the actual virtualization. Intel [24, vol. 3C] designed its virtualization extensions to be transparent to a guest system. Nevertheless, researchers suggested different kinds of side channels to detect a VM. In addition, several researchers discussed the possibility to exploit timing discrepancies to find out whether a system is running inside a VM [19, 48, 54, 55]. As virtualization adds the additional hypervisor layer beneath the already running system, certain events must be intercepted to stay in control over the VM. Both the context switches and the actual handling of these events introduce a runtime overhead which can, theoretically at least, be detected. As VMX provides the possibility to forge internal clocks of the guest, this overhead can be concealed, however. Moreover, George [19] mentioned that a guest could rely on external timers, but these are often inaccurate and require the usage of additional protection to prevent them from being forged. Since virtualization becomes more prevalent, and HyperLeech could possibly attack already virtualized targets via nested virtualization (although currently not being implemented), relying on the detection of virtualization might not be sufficient in the future anyway.

5 Related Work

Initially presented with Tribble [7] and Copilot [44], various projects used DMA to access a system's memory over protocols like PCI, PCIe, or FireWire [67]. However, several researchers showed that DMA-based approaches suffer from concurrency issues and a lack of context information about a target's execution [21, 25]. This prohibits security related tasks like tracing, debugging, and control flow analysis. Frisk [15] extends common approaches, injecting kernel implants which execute custom code within a target's kernel space. Dufflot et al. [11] exploited vulnerable firmware of a network card, using its DMA capabilities to take over the target host. Both, however, significantly modify the state of the target system. In contrast, HyperLeech is optimized to preserve a system's state and operate transparently with only a minimal impact on the analyzed target.

King et al. [28] were the first to use software emulation to stealthily virtualize a victim system. However, with the advent of hardware-supported virtualization, rootkits were able to shift already running systems into VMs without requiring a reboot. While Rutkowska [52] targeted Windows

Vista kernels, Zovi [69] attacked MacOS systems. Recently, Palutke and Freiling [42] adapted this approach attacking Linux systems, and enhanced it by locating their rootkit in *hidden memory*, certain address ranges that are not even visible to the operating system's kernel. Hypervisors have not only been used for offensive purposes, however. Approaches like [29, 39, 49, 65] perform live forensics, atomically extracting information from a guest's memory via VMI [18]. While some of these approaches require the target system to be booted inside a VM, others virtualize a system at runtime [46]. Further projects instrument a guest by injecting breakpoints into its memory, and hide these modifications by exploiting TLB incoherencies [9, 34, 47]. Fattori et al. [13] introduced an on-the-fly hypervisor that serves analysts as debugger for guest operating systems. Moreover, several approaches perform stealthy system tracing [10, 45, 51, 57, 63]. Korkin and Tanda [30] present methods to transparently control memory accesses from a hypervisor. Nguyen et al. [41] analyze malware, using a lightweight virtualized execution environment, while Rhee et al. [50] rely on a hypervisor to protect kernel structures from rootkits. Sharif et al. [56] bridged the semantic gap between the hypervisor and its guest by transparently performing malware analysis from inside the guest. In contrast, Jiang et al. [26] shift the analysis techniques outside of the guest, overcoming the semantic gap by systematically reconstructing internal semantic views. Yan et al. [64] combined hardware virtualization and software emulation to comprehend malware actions. To reveal malicious processes hidden by rootkits, both Jones et al. [27] and Litty et al. [35] suggested hypervisor-based methods to detect hidden processes by reconstructing guest information. While HyperLeech currently serves as a stealthy execution layer only, all of the above mentioned approaches might be adapted to our system. Existing approaches either need to be deployed via kernel drivers, inferring significant changes to the target system, or require the target to be booted inside a VM. While loading a kernel driver implies root privileges and support from the target kernel, booting the target inside a VM excludes systems that already run on bare metal. Besides, custom kernels might prohibit to load kernel drivers altogether. HyperLeech stealthily injects a hypervisor through DMA, allowing the virtualization of a target without the necessity to deploy a driver or to possess root privileges. In addition, the installation of our system cannot be easily inhibited by malware or intrusion prevention systems that monitor the loading of new kernel components. This is especially useful for forensic approaches, as these often need to be deployed after a system has potentially been compromised.

6 Conclusion and Future Work

To counter sophisticated anti-forensic approaches, the transparent analysis of a potentially compromised system became increasingly important. With this paper, we presented a novel

method which uses DMA provided by a PCILeech device to inject a hypervisor into a running system's volatile memory without requiring access privileges. With negligible impact on processor and memory state, HyperLeech is capable of transparently virtualizing modern multi-core Linux hosts, serving analysts as a stealthy and privileged execution layer. Compared to our approach, others rely either on virtualization based on the loading of a kernel extension, causing severely more state modifications, or suffer from a loss of context information and atomicity, being restricted to DMA. As most of today's systems do not offer appropriate protection against DMA from external devices, we expect HyperLeech to be functional on a wide variety of machines. In conclusion, our approach advances modern system analysis and memory forensics, enabling investigators to achieve sound results even in compromised environments. In the following, we point out further research directions for enhancing our current system.

Due to the configuration of EPTs, our hypervisor isolates itself from the target guest. However, EPTs can only restrict conventional memory accesses issued by the memory controller, and can be bypassed via DMA. As our hypervisor is placed within a memory region that was vicariously allocated by the target kernel, the guest should never access this area by accident. However, the target might intentionally issue DMA operations to scan its own memory for conspicuous traces. To protect the hypervisor from DMA, HyperLeech must properly configure an IOMMU, using Intel's *Virtualization Technology for Directed I/O* (VT-d). This way, the hypervisor would be fully protected from both conventional memory accesses (via EPTs) and DMA (via the IOMMU).

Attaching the PCIe Screamer device to a target host introduces a notable impact on the target state (see Section 4.1). To avoid unintended modifications, already existing management co-processors like Intel's *Management Engine* (ME) [23] or a *Baseboard Management Controller* (BMC) could be used instead. These co-processors are typically used to execute software that controls and monitors the actual host. Although mostly being signed and protected, researchers showed various ways to deploy custom modified code to run on such platforms [12, 43]. Furthermore, the open-source implementation OpenBMC could be adapted to run custom code on a BMC without requiring to exploit a vulnerability [14]. Usually, these co-processors provide their own DMA engines enabling access to the host memory for efficient data exchanges. Recently, Latzo et al. [33] presented a patch for OpenBMC running on ASPEED's *AST2500 System-on-Chip* (SoC), using it as a PCILeech device. From the host's perspective, the SoC appears as an arbitrary graphics card that connects over PCIe. This could allow the injection of the HyperLeech system without the necessity to attach additional hardware. Therefore, this would prevent the target system from detecting modifications caused by the PCIe enumeration. As a result, analysts could acquire data in a completely sound way while seemingly having no impact on the target at all.

Furthermore, virtualized targets should be able to launch their own VMs. This requires our hypervisor to provide nested virtualization, as VMX allows only one hypervisor to run in VMX root mode. Even in case the target already runs a hypervisor, our injection could be adapted to take over by withdrawing control from certain routines that are repeatedly executed in VMX root mode. Consequently, even virtualization-based rootkits could not prevent our system from being deployed unless they correctly configure an IOMMU, which so far has neither been seen in the wild nor in academia. Support for additional target operating systems and other platforms like AMD and ARM is considered hereafter.

Eventually, HyperLeech should undergo a deeper evaluation against other approaches to make a measurable statement of its advantages regarding the analysis of environment-aware malware from a forensics perspective.

Acknowledgements

We would like to thank Ulf Frisk for his helpful comments and detailed insights into the PCILeech project. Furthermore, we thank Tobias Latzo for sharing his knowledge about co-processor-based injections.

Availability

As part of this project, we make our prototype implementation available upon request for research purposes.

References

- [1] Ftdi drivers. http://www.ftdichip.com/Drivers/D3XX/FTD3XXLibrary_v1.2.0.6.zip, 2018.
- [2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 2–13. ACM, 2006.
- [3] Ramtin Amin and Ulf Frisk. PcilLeech. <https://github.com/ufrisk/pcileech-fpga/tree/master/pciescreamer>, 2019.
- [4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.
- [5] Darren Bilby. Low down and dirty: anti-forensic rootkits. *Proceedings of Ruxcon*, 2006.
- [6] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.
- [7] Brian D Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.
- [8] Jean-Christophe Delaunay. Practical dma attack on windows 10. <https://www.synacktiv.com/posts/pentest/practical-dma-attack-on-windows-10.html>, 2018.
- [9] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. SPIDER: stealthy binary program instrumentation and debugging via hardware virtualization. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 289–298, 2013.
- [10] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 51–62, 2008.
- [11] Loïc Dufлот, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, pages 378–397, 2011.
- [12] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe*, 2017.
- [13] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 417–426, 2010.
- [14] Linux Foundation. Openbmc - github. <https://github.com/openbmc/openbmc>, 2018.
- [15] Ulf Frisk. Direct memory attack the kernel. *Proceedings of DEFCON*, 24, 2016.
- [16] Ulf Frisk. PcilLeech on macos. <https://github.com/ufrisk/pcileech/wiki/Target-macOS>, 2018.
- [17] Ulf Frisk. PcilLeech on linux. <https://github.com/ufrisk/pcileech/wiki/PCILeech-on-Linux>, 2019.
- [18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed*

System Security Symposium, NDSS 2003, San Diego, California, USA, 2003.

- [19] Ou George. Detecting the blue pill hypervisor rootkit is possible but not trivial. <https://www.zdnet.com/article/detecting-the-blue-pill-hypervisor-rootkit-is-possible-but-not-trivial>, 2006.
- [20] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3): 362–368, 2011.
- [21] Michael Gruhn and Felix C Freiling. Evaluating atomicity, and integrity of correct memory acquisition methods. *Digital Investigation*, 16:S1–S10, 2016.
- [22] Takahiro Haruyama and Hiroshi Suzuki. One-byte modification for breaking memory forensic analysis. *Black Hat Europe*, 2012.
- [23] Intel. Intel management engine. <https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html>, 2017.
- [24] Intel. Intel 64 and ia-32 architectures software developer’s manual volume 3 (3a, 3b, 3c & 3d): System programming guide, part 3. *Part*, 3, 2019.
- [25] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 605–620, 2014.
- [26] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.
- [27] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 91–100, 2008.
- [28] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 314–327, 2006. doi: 10.1109/SP.2006.38. URL <https://doi.org/10.1109/SP.2006.38>.
- [29] Michael Kiperberg, Roe Leon, Amit Resh, Asaf Al-gawi, and Nezer Zaidenberg. Hypervisor-assisted atomic memory acquisition in modern systems. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy, ICISPP 2019, Prague, Czech Republic, February 23-25, 2019*, pages 155–162, 2019.
- [30] Igor Korokin and Satoshi Tanda. Detect kernel-mode rootkits via real time logging & controlling memory access. *CoRR*, abs/1705.06784, 2017.
- [31] Nisha Lalwani, MB Chandak, and RV Dharaskar. Split personality malware: a security threat. In *IJCA Proc. National Conf. Innovative Paradigms in Engineering and Technology (NCIPET 2012)*, number 14, 2012.
- [32] Tobias Latzo, Ralph Palutke, and Felix C. Freiling. A universal taxonomy and survey of forensic memory acquisition techniques. *Digital Investigation*, 28 (Supplement):56–69, 2019.
- [33] Tobias Latzo, Julian Brost, and Felix Freiling. Bmcleech: Introducing stealthy memory forensics to bmc. *Digital Investigation*, 2020.
- [34] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiyayas. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 386–395, 2014.
- [35] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 243–258. USENIX Association, 2008.
- [36] Carsten Maartmann-Moe. Inception. <http://www.breaknenter.org/projects/inception/>, 2011.
- [37] A. Theodore Marketos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.
- [38] Alex Markuze, Adam Morrison, and Dan Tsafir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016, pages 249–262, 2016.

- [39] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*, pages 297–316, 2010.
- [40] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. IOMMU protection against I/O attacks: a vulnerability and a proof of concept. *J. Braz. Comp. Soc.*, 24(1):2:1–2:11, 2018.
- [41] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. MAVMM: lightweight and purpose built VMM for malware analysis. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, USA, 7-11 December 2009*, pages 441–450, 2009.
- [42] Ralph Palutke and Felix C. Freiling. Styx: Countering robust memory acquisition. *Digital Investigation*, 24: S18–S28, 2018.
- [43] Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its bmc: the hpe ilo4 case. *Recon Brussels*, 2018.
- [44] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.
- [45] Jonas Pfoh, Christian A. Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*, pages 96–112, 2011.
- [46] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. Follow the whiterabbit: Towards consolidation of on-the-fly virtualization and virtual machine introspection. In *ICT Systems Security and Privacy Protection - 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings*, pages 263–277, 2018.
- [47] Sergej Proskurin, Tamas K. Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the shadows: Empowering ARM for stealthy virtual machine introspection. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 407–417, 2018.
- [48] T Ptacek, Nate Lawson, and P Ferrie. Don't tell joanna, the virtualized rootkit is dead. *Black Hat*, 2007.
- [49] Zhengwei Qi, Chengcheng Xiang, Ruhui Ma, Jian Li, Haibing Guan, and David S. L. Wei. Forevisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Trans. Cloud Computing*, 5(3):443–456, 2017.
- [50] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*, pages 74–81. IEEE Computer Society, 2009.
- [51] Paul Royal. Alternative medicine: The malware analyst's blue pill. *Black Hat USA*, 2008.
- [52] Joanna Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.
- [53] Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. *Proceedings of BlackHat DC*, 2007, 2007.
- [54] Joanna Rutkowska. Security challenges in virtualized environments. In *Proceedings RSA conference 2008*, 2008.
- [55] Joanna Rutkowska and Alexander Tereshkin. Is-gameover () anyone. *Black Hat, USA*, 2007.
- [56] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 477–487, 2009.
- [57] Jianguyong Shi, Yuexiang Yang, Chengye Li, and Xiaolei Wang. SPEMS: A stealthy and practical execution monitoring system based on VMI. In *Cloud Computing and Security - First International Conference, ICCCS 2015, Nanjing, China, August 13-15, 2015. Revised Selected Papers*, pages 380–389, 2015.
- [58] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63): 504–533, 2005.
- [59] Johannes Stüttgen and Michael Cohen. Anti-forensic resilient memory acquisition. *Digital investigation*, 10: S105–S115, 2013.

- [60] Johannes Stüttgen and Michael Cohen. Robust linux memory acquisition with minimal target impact. *Digital Investigation*, 11(1):S112–S119, 2014.
- [61] Jacob Torrey. More shadow walker: Tlb-splitting on modern x86. *Blackhat USA*, 2014.
- [62] Stefan Vömel and Felix C Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3–22, 2011.
- [63] Carsten Willems, Ralf Hund, and Thorsten Holz. Cx-pinspector: Hypervisor-based, hardware-assisted system monitoring. *Ruhr-Universität Bochum, Tech. Rep.*, page 12, 2013.
- [64] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. *ACM Sigplan Notices*, 47(7):227–238, 2012.
- [65] Miao Yu, Zhengwei Qi, Qian Lin, Xianming Zhong, Bingyu Li, and Haibing Guan. Vis: Virtualization enhanced live forensics acquisition for native system. *Digital Investigation*, 9(1):22–33, 2012.
- [66] Patrycjusz Zdzichowski, Michal Sadlon, Teemu Uolevi Väisänen, Alvaro Botas Munoz, and Karina Filipczak. Anti-forensic study. *NATO CCDCOE (NATO Cooperative Cyber Defence Centre of Excellence)*, 2015.
- [67] Lei Zhang, Lianhai Wang, Ruichao Zhang, Shuhui Zhang, and Yang Zhou. Live memory acquisition through firewire. In *Forensics in Telecommunications, Information, and Multimedia - Third International ICST Conference, e-Forensics 2010, Shanghai, China, November 11-12, 2010, Revised Selected Papers*, pages 159–167, 2010.
- [68] Ning Zhang, Kun Sun, Wenjing Lou, Yiwei Thomas Hou, and Sushil Jajodia. Now you see me: Hide and seek in physical address space. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 321–331, 2015.
- [69] Dino A Dai Zovi. Hardware virtualization rootkits. *Black Hat 2006, August, 2006*.

Effective Detection of Credential Thefts from Windows Memory: Learning Access Behaviours to Local Security Authority Subsystem Service

Patrick Ah-Fat
Imperial College London

Michael Huth
Imperial College London

Rob Mead
Microsoft

Tim Burrell
Microsoft

Joshua Neil
Microsoft

Abstract

Malicious actors that have already penetrated an enterprise network will exploit access to launch attacks within that network. Credential theft is a common preparatory action for such attacks, as it enables privilege escalation or lateral movement. Elaborate techniques for extracting credentials from Windows memory have been developed by actors with advanced capabilities. The state of the art in identifying the use of such techniques is based on malware detection, which can only alert on the presence of specific executable files that are known to perform such techniques. Therefore, actors can bypass detection of credential theft by evading the static detection of malicious code. In contrast, our work focuses directly on the memory read access behaviour to the process that enforces the system security policy. We use machine learning techniques driven by data from real enterprise networks to classify memory read behaviours as malicious or benign. As we show that Mimikatz is a popular tool seen across Microsoft Defender Advanced Threat Protection (MDATP) to steal credentials, our aim is to develop a generic model that detects the techniques it employs. Our classifier is based on novel features of memory read events and the characterisation of three popular techniques for credential theft. We integrated this classifier in a detector that is now running in production and is protecting customers of MDATP. Our experiments demonstrate that this detector has excellent false negative and false positive rates, and does alert on true positives that previous detectors were unable to identify.

1 Introduction

The past ten years have seen a dramatic increase in the number and sophistication of cyber-attacks. These changes have as a consequence that the devices and networks of private citizens, businesses, and government agencies can often no longer be protected against all types of such threats. That malicious actors have already infiltrated a network of computers and devices is therefore nowadays a generally accepted wisdom.

This assumption gives great importance to the detection of suspicious or malicious activity that happens within a network perimeter, so that security measures can be taken to contain such activities and the damage they may cause. Modern enterprise security systems thus must ally protection with detection in order to be able to expose intruders that have managed to defeat protection mechanisms and, in doing so, have already compromised the security of the network.

Our research in intrusion detection, like that of others [2, 9, 11], therefore assumes that an attacker has already compromised a network through some vulnerability such as a security-critical bug in system software. For effective intrusion detection we therefore need reliable tools to identify actions within the network that very likely stem from an intruder who has malicious aims.

The development of such tools will benefit from models of intent for attacks. For example, if the intent of an attacker is to steal monetary assets from a bank network, this gives us an idea of a set and possible sequence of actions an attacker would take in that network. Conversely, the identification of malicious actions that were taken can help a security monitor with understanding dynamically the intent of an attacker.

Some of these actions are building blocks of many different types of malicious intent. Detecting actions that are shared in many attacks is therefore particularly valuable for identifying that an actor who already penetrated a network is conducting an attack from the inside; it also provides flexibility in identifying behaviours that are common to a variety of attacks.

In this paper, we focus on one particular such action known as credential theft [3, 6, 14, 17–19]. It is well known that credential theft is a very common behaviour in attackers who have compromised a machine. Indeed, credentials are sensitive information that are valuable since their possession may constitute a means of privilege escalation or lateral movement: an attacker who has managed to steal security-relevant credentials on a machine within that network could use these credentials, for example, to get administrator rights on a targeted machine, access rights to other machines on the network, and so forth. It is therefore hardly surprising that a

large variety of methods have been developed for stealing credentials [4, 5], including phishing, keylogging, password spraying, and wireless traffic monitoring.

Our work reported in this paper investigates a specific, elaborate technique that advanced attackers, and computer forensic analysts, have developed in order to retrieve users' credentials. This technique repeatedly reads from machine memory in search of credentials [8]. Our focus will be on the application of that technique within the Windows operating system, although our contributions may well be transferable to other operating systems and their security processes and monitoring environments. We made this choice because:

- Windows is an important operating system used widely in private, commercial, and government networks, and
- we have the ability to promote the transfer of salient outcomes of this work into Windows products.

Given our focus on Windows and its Security Architecture, we concentrate on the detection of malicious memory accesses to a particularly sensitive process, the *Local Security Authority Subsystem Service* (LSASS). This critical system process enforces security policies on Windows machines.

LSASS is responsible for performing essential tasks related to security such as verifying user logins, managing passwords, and creating access tokens. As such, this process contains a considerable amount of very sensitive data in its memory, making LSASS a prime target on Windows to any internal attacker that seeks to steal credentials.

For our approach, it is important to note that not only attackers with malicious intent will run a process that reads from LSASS memory. Other processes may read from LSASS memory with perfectly good intent, and perform such reads for the benefit and health of the overall network.

In fact, many anti-virus and security software products for Windows scan the LSASS process memory during daily routine checks in order to detect any potentially corrupted or infected files. Such benign reads widely occur on Windows networks. Indeed, on average over all data provided by Microsoft Defender Advanced Threat Protection (MDATP), the LSASS memory is read 40 million times and by many thousand different processes a day.

Our work therefore faces the problem that memory reads to LSASS can be either malicious or benign. The aim of this research is therefore to develop a method that can decide whether a process that is reading from LSASS memory is simply performing a benign action, for example within the context of a virus scan, or whether it is actually trying to steal sensitive information, notably credentials.

The nature of this problem suggests the use of techniques from machine learning, those that can build models for classification. In our setting, we particularly seek models for binary classification that can judge whether a process that is reading from LSASS memory is malicious or not.

We determined a specific methodology in order to solve this classification problem in a manner that would allow us to

deploy its solution in a Windows product. That methodology and the contributions made in its execution are described next.

Methodology and Contributions of paper: Our adopted methodology and contributions are as follows:

1. We use real-world data to determine novel features and a model for a precise analysis, control, and characterisation of LSASS memory access.
2. We demonstrate that these features allow us to compute “signatures” of processes, determined by how these processes access LSASS memory.
3. We further show that different sessions of the same process will tend to aggregate into a single linearly identifiable cluster of such read behaviour.
4. We harvest malicious read accesses to LSASS memory by collecting the reads performed by known penetration testing tools run on MDATP customers' machines.
5. We characterise this harvested data more precisely via a set of linear regressions that enable us to distinguish benign LSASS read accesses from malicious ones.
6. Based on this, we design an effective detector that is able to identify suspicious read accesses to LSASS memory.
7. We experimentally confirm that this detector has low false positive and false negative rates. Also, our detector highlights some interesting instances of true positives.
8. Finally, we deployed this detector in production on MDATP, where it now actively protects customers.

These outcomes of our work also have good potential to be integrated with online detection tools, so that the latter can become more effective. For example, the machine-learning models of the detector may be updated based on daily statistical data collected in Windows systems and enriched with security-relevant information from external sources such as cyber-intelligence or cyber-threats centres. This may also lead to the design of non-linear classifiers in future work.

Outline of paper: Related work is reviewed in Section 2. We detail our methodology in Section 3. Section 4 introduces our novel model for memory reads. Patterns in benign read behaviour are explored in Section 5. The harvest and analysis of malicious memory read behaviour is described in Section 6. We build and evaluate our detector in Section 7 and highlight interesting alerts it produces in Section 8. Section 9 studies how our approach deals with Windows updates. Section 10 considers the resiliency of our detector against adversarial manipulation. We suggest future work in Section 11, discuss our work further in Section 12 and conclude in Section 13.

2 Related Work

2.1 Credential Theft by Memory Dumping

An important way for intruders to retrieve credentials is to read from the LSASS memory. In order to do so, attackers

are required to have the debug privilege to gain access to this protected memory location. Since we assume that the network has been infiltrated by malicious actors, we will also assume that attackers have already gained the debug privilege and are trying to access memory maliciously.

Attackers have designed a variety of tools [12] that can read the memory of the LSASS process on Windows machines to extract credentials – notably, LsIsass, Windows Credential Editor, and Mimikatz. Otherwise benign processes, e.g. `procdump` or `taskmgr`, can be abused to perform living-off-the-land attacks by dumping the whole of the LSASS process memory. Such credential theft techniques have been used in large scale cyber attacks, e.g. NotPetya and Olympic Destroyer [13].

As discussed in Section 3, Mimikatz [1] is a common tool of choice for attackers that want to steal Windows credentials, and is the most prevalent of the tools mentioned above in MDATP machines world-wide in 7 months.

Mimikatz offers a variety of Windows system techniques in order to extract sensitive information. These different credential-access techniques can be chosen by passing command line arguments to the Mimikatz executable. Mimikatz can also be launched in an interactive mode without the use of any command line arguments, or it may be invoked remotely via PowerShell – a Windows command-line shell designed for system administrators.

For our work, it makes sense to study the three most popular techniques that can steal credentials from the LSASS memory of a targeted machine. These three techniques are:

- L1 Enumerate logon passwords, with command line argument `sekurlsa::logonpasswords`
- L2 Steal Kerberos tickets, with command line argument `sekurlsa::tickets`
- L3 Pass the hash, with the following command line argument `sekurlsa::pth`

2.2 Memory Dumping Detection

The current detection mechanisms for such credential dumping activity are mainly static [15, 16]. Commonly used anti-virus software is able to detect the presence of the executable file `mimikatz.exe` and contains this threat by quarantining it.

Yara rules are used to detect a malicious software by looking for characteristics or pattern in an executable file. A specific set of Yara rules has been written by Benjamin Delpy, the author of Mimikatz, to recognise Mimikatz executable file. These rules are able to prevent the process Mimikatz from running on a machine. However, these rules cannot be used to prevent attackers from running Mimikatz through remote execution via PowerShell or through process injection. Attackers could also bypass those basic security measures by renaming and recompiling their own custom version of Mimikatz in order to circumvent those detection mechanisms.

Other detection mechanisms are based on the use of so called *honey hashes* that are meant to be stolen and to raise

an alert at the time they are reused [7]. However, reliance on that mechanism could leave customers unprotected since this would not detect whenever an attacker steals credentials and retains them for future use in a more elaborate attack.

In contrast, the aim of our work reported in this paper is to analyse and detect the *actual behaviour* that a process exhibits when accessing LSASS memory. Our approach therefore directly aims at detecting malicious behaviour rather than a static proxy for a malicious actor. Our approach is therefore also resilient to the aforementioned countermeasures that an attacker may adopt: process injection and remote execution. This is so since our approach classifies actual behaviour of reads, rather than detecting the presence of programs that are known to facilitate such behaviours.

2.3 LSASS architecture

The Local Security Authority Subsystem Service (LSASS) is a system process present on Microsoft Windows operating systems. Its high level roles include providing services to authenticate to the local computer and domain as well as maintaining information on aspects of security on a machine.

LSASS stores credentials in memory on behalf of users with an active session. These cached credentials allow users to access other resources on a Windows domain that are secured with the same identity, without the user having to re-enter a password every time access is required.

These credentials can take multiple forms, including hashes (NT and LAN Manager) and Kerberos tickets. Credentials are cached locally inside LSASS process memory when a user performs an authenticated action on that machine, such as logging on using Remote Desktop, scheduling a task or executing a process using 'RunAs'.

A handle with `PROCESS_VM_READ` can be opened on the LSASS process object and leveraged to perform a cross-process read of sensitive data. To do this the Security Descriptor of the LSASS process object must explicitly grant the accessor this right. Alternatively the accessor can hold `SeDebugPrivilege`; then access to all non-protected processes is granted regardless of their Security Descriptor.

`SeDebugPrivilege` can be acquired by any member of the Local Administrator group, therefore allowing any local administrator to read the memory of the LSASS process. This technique can be used maliciously to extract from LSASS the cached credentials of other users on a Windows domain. These users may hold higher levels of privilege or access to special resources that the current local administrator does not.

Over the last several years technologies have been introduced to the Windows operating system to prevent this attack vector, such as LSASS as Protected Process Light (PPL) and Virtualisation Based Security (VBS). These works do either prevent access to the LSASS process from a Local Administrator or move sensitive data out of the process entirely. However these technologies are not always enabled on a sys-

tem, as enabling them can present compatibility issues with custom or third party software – and in the case of VBS – can require certain hardware which is not always available. Therefore these technologies are not enabled in all customer environments, leaving cross-process reads of LSASS process memory a workable technique to elevate privilege on a Windows domain that can be leveraged by malicious actors. Our approach to detection thus becomes of particular interest in the cases where PPL and VBS cannot be enabled.

2.4 Learning Intruder Behaviours

Previous works have focused on designing detection mechanisms by learning the behaviour of legitimate and malicious agents. The work in [20] focuses on intrusion detection by learning patterns in system call sequences. It aims at being generic and at identifying different kinds of intrusion such as buffer overflows or denial of service attacks. In [10], the authors focus on analysing system call traces produced by the `sendmail` program and they propose ways of learning rules for detecting normal and abnormal sequences. Other works, which focus on university account theft [22], analyse authentication logs and build on heuristics to introduce features that are sensible to the situation such as temporal-spatial violation or inconsistencies based on resource usage. More specific works have focused on intrusion detection on Android devices [21]. Their machine learning mechanisms rely on features based on dataflow-related APIs which are particularly suited for describing Android application behaviours.

Although these works aim at detecting different kinds of intrusion, our work aims at building precise methods for identifying a specific and particularly common action that intruders may perform which is credential theft from Windows memory. All of those works resorted to heuristics based on security knowledge and domain expertise in order to build features that were suited to analyse their specific situation and that made machine learning algorithms exploitable. Similarly, as our work means to study the memory read behaviours of different programs, we introduce in Section 7 the features that we used in our analyses and we justify our intuition as for why they are meaningful in our scenario.

3 Methodology Driven By Customer Data

We now develop and justify our approach and its methodology, which are motivated by two chief objectives:

- Develop an accurate model for read behaviour of LSASS memory as a basis for classifying malicious reads.
- Protect customers of MDATP against current attacks that steal credentials from LSASS memory through detectors that use such accurate models.

As we aim at protecting MDATP customers, we place great value on the data that originates from machines of genuine MDATP customers, rather than relying on data generated by

simulations on test machines. This rationale justifies and shapes the manner in which we collect our data and perform our subsequent analyses. These data are collected as part of an opt-in, commercial relationship between Microsoft and their enterprise customers. The data are collected from individual operating systems as part of the MDATP deployment, in nearly real time, and sent to Azure to provide detection and remediation capabilities. Care is taken to obfuscate personal information before presentation to researchers.

We analysed the processes that were reading from the LSASS memory on all machines that are customers of MDATP and looked for credential dumping tools such as `Lslass`, `Windows Credential Editor` and `Mimikatz`. The only tool that we effectively observed running in a non-obfuscated manner on customers' machines was `Mimikatz`.

We thus decided to study the read behaviour of LSASS memory for a set of prominent anti-virus and security software tools, but also for `Mimikatz` as the most relevant tool an attacker appears to choose. As explained further in Section 4, our intuition for building a model of such read behaviours came from studying the way in which such credential dumping tools work, with a particular emphasis on `Mimikatz`.

Our data collection and machine learning then proceeded in the following stages:

- We analysed the behaviour of the most prevalent benign software that scan the memory of the LSASS process by examining a random sample of seventy-thousand different machines that run MDATP.
- We then harvested the LSASS read behaviour of credential theft techniques performed by `Mimikatz` seen on real customers' machines by harvesting such attacks on 244 different machines that run MDATP.
- Based on these data, we trained and tested a detector for credential theft.
- Finally, we analysed the influence of Windows update on the read behaviour of credential theft techniques based on recent data, by collecting and comparing malicious read behaviours on machines running on different Windows update versions.

In Section 11, we discuss potential countermeasures that an attacker could devise in order to bypass our detector. Regardless of bypass techniques, however, the practical benefits of our detector were deemed powerful enough to lead to their integration into the deployed MDATP detection suite.

Our methodology also has the advantage of being rather generic. Therefore, this novel way of characterising malicious processes may be transferable to other kinds of malicious behaviour or may safeguard other parts of the Windows memory.

4 Modelling and Collecting Memory Reads

LSASS contains secrets that are stored in its process memory. Keys and credentials held here are attractive to attackers as they enable them to perform operations such as decrypting

sensitive data, or impersonating users that have a higher level of privilege on the domain.

Tools that read from LSASS memory in order to obtain these secrets, such as Mimikatz, perform a number of cross process reads of the address space of the LSASS process – using Windows API calls such as `ReadProcessMemory`.

These APIs take an address of a target process and size of the portion to be read as their arguments. Assuming the caller has the required privileges to read from the target, the data from this address is then copied to a supplied buffer located in the caller process.

The address space layout of the LSASS process is dynamic and varies depending upon a number of factors, such as Address Space Layout Randomisation (ASLR), the timing of memory allocations, the type and size of data that is stored, and on how long the process has been running. It is therefore extremely difficult to predict the address at which secrets of interest reside within the LSASS memory.

Since one cannot simply provide an address and perform a single read to obtain the required information, tools such as Mimikatz must instead first perform multiple reads of the LSASS process in order to search for addresses at which the sensitive data resides.

Whilst the target process address space is generally unpredictable, there are some fixed frames of reference that can be exploited in this activity. For example, the address of the Process Environment Block (PEB) of the remote process can be obtained by an API call such as `NtQueryInformationProcess`. The target Process Environment Block can then be read to discover the location of certain modules loaded by the process.

Once these module locations are known, pointers from structures that are stored as global variables inside these modules can be read predictably, where this predictability varies slightly with the operating system versions. When these pointers are followed, they lead to other structures that can act as clues that ultimately help to unravel and reveal the address where credentials or other secrets are stored.

For example, the Mimikatz `sekurlsa::logonpasswords` operation enumerates and displays all credentials for logon sessions stored within LSASS and follows this approach:

- It performs multiple remote reads against the LSASS Process Environment Block to retrieve loaded module information for loaded `lsa` package `dll` files.
- For the `MSV1_0` authentication package, it remotely reads all of `msv1_0.dll` into a local buffer. Starting at the base address, it searches for a series of *magic bytes* that are located in close proximity to a pointer to the logon session list.
- It mines a series of magic bytes that are in close proximity to an AES key and Initialisation Vector (IV) located in LSASS memory and remotely reads the key and vector to a local buffer.
- It copies the size of the logon session list out of LSASS

memory.

- For each of the items in the session list, it remotely reads various details from the session out of LSASS memory in individual reads – username, domain, SID, etc., and a pointer to where the credentials are stored.
- A number of reads from LSASS are then performed to pull different structures out of LSASS, the last one contains encrypted credentials as a unicode string.
- The encrypted credential material is then decrypted using the previously retrieved key and IV.

These operations result, over short time periods, in a large and varying number of reads of the LSASS process memory.

Since Windows 10 RS3, insights into memory reads of target processes are possible – due to instrumentation of the memory manager. Telemetry from this instrumentation is available to security vendors via the Microsoft-Windows-Threat-Intelligence Event Tracing for Windows provider.

This telemetry information provides details of the calling process, the target process, and the size of the data that was copied. It is not practical to collect all of these events. Even if we were to restrict the collection of these events to the target of LSASS, the volume of data generated from a single host would be too large to be practical at network scale.

Instead, aggregates of this data are created by MDATP. These aggregates summarise data into 5-minute time slices, which we will refer to as *sessions*, that provide the calling process, target process, the total of bytes read and the number of reads that occurred within that time slice.

Given the total number of bytes read and the number of reads within that time period, it seems natural to model the behaviour of credential theft tooling such as Mimikatz by these data aggregates understood as features for machine learning. We conjecture that the manner in which malicious tools extract secrets from LSASS memory is different enough from the read behaviour of LSASS memory for benign tools, when we understand behaviour in terms of those features.

The next sections will report on our experimental work for testing and confirming this conjecture.

5 Patterns in Benign Read Behaviours

Based on the model of behaviour and the features of such behaviour introduced in the previous section, we next investigate the behaviour of LSASS memory reads for benign processes that run anti-virus or other security software. This investigation means to identify whether benign read behaviour follows any particular statistical patterns that could differentiate them from processes that run tools which aim to steal credentials.

To that end, we performed the following experiment:

Experiment 1. *We collected 100k read sessions randomly sampled over 10 days on 70k different machines running MDATP. We plot in Figure 1 the behaviour of 5 of the most prevalent processes to read memory from LSASS: WmiPrvSE,*

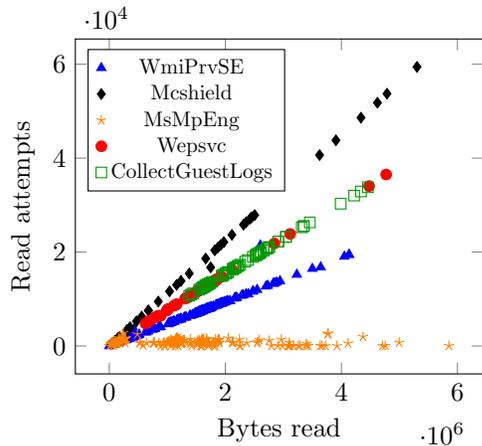


Figure 1: Read behaviour of LSASS memory for the 5 most prevalent security software and anti-virus applications, with data collected over 10 days across 70k different machines. Units are millions (x-axis) ten-thousands (y-axis).

Mcshield, MsMpEng, Wepsvc and CollectGuestLogs which are security software tools which scan the LSASS memory for potentially infected files. Recall that for each of those 5-minute sessions performed by those tools, we recorded the number of bytes read and the total number of read attempts performed during that session. Figure 1 shows the read sessions whose number of bytes read was between 0 and 6 million. We chose this range as it will be of particular interest when comparing benign with malicious behaviours in Section 6.

The plots in Figure 1 suggest that data are very sparse, and that the large majority of the data points appear to aggregate on four different highly linear relationships passing through the origin. For each of those four clusters, we could fit a linear model for classification in terms of the selected features. Some of these linear clusters contain a single process, such as the one formed by `WmiPrvSE`. This suggests that it is reasonable to devise a method for identifying whether a process reading from LSASS memory is indeed `WmiPrvSE` or not – based on the data point that it leaves on this graph.

The processes such as `CollectGuestLogs` and `Wepsvc` seem to lie on the same line, and it would thus be more difficult to distinguish both processes by looking at their read behaviours of LSASS memory, especially for those data points that could represent either of those processes.

However, it is perfectly reasonable that two security software applications may perform similar routine sanity checks on the same area of memory, and therefore may have similar read behaviour of LSASS memory. Fortunately, we have no need to distinguish between different benign processes. At the abstract level of classification of malicious read behaviour of LSASS memory, it is safe to identify such benign processes if their behaviour is sufficiently similar.

Also, the fact that the benign processes we are studying are

meant to perform similar tasks might explain why the data that we observe for them is localised into specific clusters.

Our conjecture suggests to study the read behaviour of LSASS memory for malicious processes that are trying to steal credentials, and to determine whether the patterns of such malicious reads are sufficiently different from those of benign read behaviours. This is the topic of the next section.

The classification and the resulting detector that we develop in this paper have uses beyond the mere classification into benign or malicious process. For example, a process whose behaviour seems anomalous compared to all data clusters of applications that are known to be benign may be deemed to be suspicious even if no prior data about that process is available. Similarly, when the actual read behaviour of an application that is known to be benign deviates from what its data cluster would suggest, this can indicate that the application has become infected – triggering an appropriate response.

6 Harvest and Analysis of Malicious Reads

We have seen that the behavioural model of reads from LSASS memory, introduced in Section 4, provides us a useful and effective characterisation of how benign processes access LSASS memory. In this section, we analyse the read behaviour of LSASS memory deemed most important to credential theft techniques. This requires an experiment to collect data from real Windows networks, and a subsequent analysis of that data in order to assess whether our behavioural model allows us to distinguish benign from malicious processes, as claimed in our conjecture.

As already emphasised in Section 3, our research and business objective is to protect Windows customers from threats that are existing at present. Since many current credential thefts from LSASS memory are based on use of the `Mimikatz` tool, we focus our attention on studying the behaviour of that tool when it is invoked on machines of customers.

To that end, we conducted the following experiment:

Experiment 2. *We collected some data for all instances of `Mimikatz` invocations that we observed on machines that use `MDATP`, from January to July 2019. This exercise gathered a total of 1600 `Mimikatz` instances that were collected on 244 different machines within that 7-month period. Out of these 1600 instances, 256 were interactive sessions and were launched without any command line arguments. We could apply an unsupervised classification technique in order to label these data points, but for simplicity purposes we discarded those entries from our analysis. Thus, we obtained 1344 labelled data points from this experiment, instances of `Mimikatz` invocations that have one of three possible labels `L1`, `L2` or `L3` – denoting which of the three commands listed in items `L1-3` in Section 2 were used in these respective instances.*

Out of those 1344 labelled instances, 39 instances were isolated data points which we decided to discard for the purpose

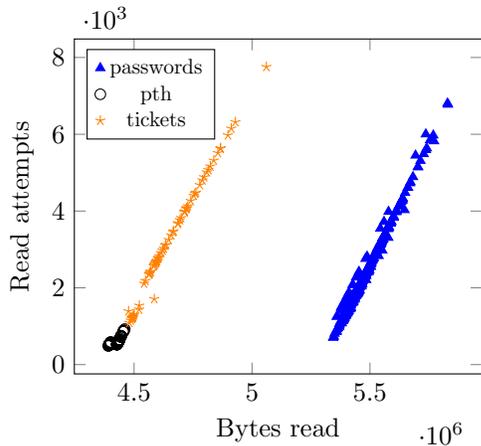


Figure 2: Read behaviour of LSASS memory under credential theft technique L1 (password), L2 (ticket), and L3 (pass the hash). The 1344 data points were collected on 244 different machines during 7 months, as described in Experiment 2.

of our analysis. Most of those discarded were data points with a particularly low number of bytes read, which we considered as failed attempts to run Mimikatz by users who possibly did not have the required privilege to do so. We were therefore left with 1305 labelled instances of Mimikatz invocations.

A plot of the resulting dataset is seen in Figure 2; data is labelled with the particular memory read technique that was used. We compare in Figure 8 in Appendix A the position of those malicious data points to those of benign read instances that were studied in the previous section. This geometric visualisation enables us to make two important observations.

First, the data in Figure 8 shows that processes that steal credentials from LSASS memory follow a read behaviour that is clearly identifiable by our behavioural model introduced in Section 4. Indeed, we can see that the data points corresponding to read access of LSASS memory by Mimikatz are easily distinguishable from those that correspond to read access of LSASS memory made by benign and legitimate security software applications, whose task is to perform regular sanity checks. The data points of malicious processes lie on a very specific and localised area of the plot. In contrast, processes of benign security software create data points that span other regions of the feature space.

This analysis therefore encouraged us to build a detection method based on the fact that malicious activity can be identified by looking at the LSASS memory read behaviour of a process. This detector is developed in the next section.

Second, from the data shown in Figure 2 we can clearly see that our behavioural model of LSASS memory reads enables us to further characterise malicious processes by distinguishing between different credential theft techniques used in such attacks. Therefore, it seems possible to develop a detector that can not only identify that malicious read behaviour of

LSASS memory takes place, but that can also classify which credential theft technique is being used in that attack. Such information is valuable as a guide for security response and to improve security intelligence data.

7 Modelling Credential Theft

The experiments and analysis of the collected data on the read behaviours of LSASS memory, described in Sections 5 and 6, suggest that it is feasible to build a detection mechanism for credential theft based on such read behaviour of processes. This section discusses how we built such a detector, based on the data collected in Section 6.

We split the cleaned dataset, containing the 1305 labelled memory reads performed by Mimikatz instances, into 3 datasets following a standard 60%-20%-20% distribution: a training set of size 783, and a validation and a test set both of size 261. In order to build this detector, we studied the different memory read techniques L1-L3 individually and we noticed that each of them was producing data that seemed to aggregate on linear clusters.

The idea of our detector is thus to consider that a data point witnesses a malicious memory read technique if it lies within a certain interval around the corresponding line. To realise this idea, we performed logistic regressions for the data points on the training set, for each of the techniques L1-L3.

We then used the validation set to analyse the influence of the width of the detection interval on the classification rates. Then, we use the test set to establish that we obtain low false positive (FP) and false negative (FN) rates.

7.1 Training

We display in Figure 3 a zoomed-in view of the data points that we collected for the memory read technique L1, which steals logon passwords from LSASS memory. We can see that those points seem to lie on a line, which confirms the effectiveness of our behavioural model. For these data, we performed a logistic regression with the least squares method based on the following model:

$$X = a \cdot Y + b + \epsilon \quad (1)$$

where X and Y represent the number of bytes read and the number of read attempts respectively, a and b are two constants that we wish to estimate and ϵ represents the error term. The obtained regression line is depicted on the plot. For all positive real α , we also define the *alert interval* \mathcal{I}_α to encompass all the points which lie within a vertical distance of α standard deviations from the regression line, i.e. all points whose error term defined in (1) verifies $|\epsilon| \leq \alpha \cdot \sigma$ where σ represents the sampled standard deviation of the errors for that regression. The graph also shows the alert interval \mathcal{I}_3 .

For the technique L2, which enumerates Kerberos tickets, a very similar plot is shown in Figure 9 in Appendix B. The

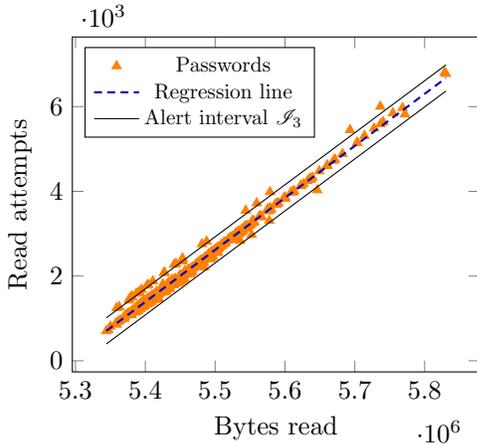


Figure 3: Memory read behaviour of passwords theft.

data acquired for the pass the hash technique (L3) is displayed in Figure 10 in Appendix B. The data points corresponding to this technique are divided into 2 different linear clusters for which we therefore performed separate logistic regressions.

We can empirically observe on those plots that the alert interval seems to include a large majority of the data points. A natural question that arises in this context is what interval width should be chosen in order to build an effective detector.

Large widths would lead to a higher true positive rate but would also increase the false positive rate. On the other hand, more narrow intervals would reduce the false positive rate but would also be more likely to miss out on some true positives.

We study the influence of this interval width on the false positive and true positive rates in the next subsection.

7.2 Validation

For validation, we study how the false positive and true positive rates vary as the margin for our detection interval varies. The aim of this study is to find an ideal trade-off between a low false positive and a high true positive rate.

For this we use a parameter α , a positive real number that is used to determine a threshold value for our detector. We build this detector as follows: for each of the techniques L1-L3, a data point is considered as belonging to that technique if it lies within the corresponding alert interval \mathcal{I}_α .

The pseudo-code for this is shown in Figure 4. In this algorithm, α is a positive real number that characterises the detector interval width. The set Γ refers to the set of the 4 linear regressions that we perform for the 3 credential theft techniques, including 2 separate ones for the pass the hash technique. For each regression τ :

- x_τ^{min} , and x_τ^{max} refer to the minimal and maximal number of bytes read for regression τ ,
- a_τ , b_τ and σ_τ (resp.) represent the slope, the y-intersect and the standard deviation of the regression residuals.

Input: Data point (x, y) modelling a memory read session

Output: None or alert on malicious memory read

```

1: for  $\tau$  in  $\Gamma$  do
2:   if  $x \in [x_\tau^{min}, x_\tau^{max}]$  then
3:     if  $|a_\tau \cdot x + b_\tau - y| < \alpha \cdot \sigma_\tau$  then
4:       Raise alert for technique  $\tau$ 
5:     end if
6:   end if
7: end for

```

Figure 4: Algorithm that detects malicious LSASS memory read behaviour, parametrised by the interval width and a hyper-parameter α for the alert threshold.

This algorithm takes as input a pair (x, y) as representation of a read session: x is the total number of bytes read, and y the total number of read attempts during the process session. For each technique, the algorithm raises an alert if the input data point lies within the respective alert interval \mathcal{I}_α .

In order to compute the true positive rate that this algorithm yields, we applied this detection on each of the $N = 261$ Mimikatz instances of the validation set, and then computed the true positive rate as n/N where n was the number of detections that correctly classified these a instances.

Experiment 3. For the computation of the false positive rate, we ran our detector on all the benign (legitimate) read events that we collected over 7 days from MDATP machines world-wide. This amounted to a total M of 273 million reads from MDATP customers' machines world-wide.

We then computed the number m of alerts that our algorithm generated for these events, and computed the false positive rate as m/M . Since the number of true positives obtained in a single day is negligible compared to the daily number of read events, the total number of benign read events can be approximated by the total number of reads. In order to compute the false positive rate, our security experts manually examined all the positive cases that our detector alerted on and excluded the true positive ones by inspection.

We also ran our detection algorithm for different values of α and recorded the values of the false positive and true positive rates for these variations, depicted in Figure 11 in Appendix C. Naturally, we can see that, as the detector interval width increases, so do the false positive and true positive rates. After the analysis of this graph, we decided to select the value $\alpha = 3$ for our detector, which seems to yield an excellent trade-off between a high 95% true positive rate and an outstanding false positive rate in the order of magnitude of 10^{-6} .

This chosen value of α corresponds to the alert interval \mathcal{I}_3 highlighted in Figures 3, 9 and 10, and we can graphically confirm that this interval covers a large majority of the data points we collected – without being too wide.

Technique	Passwords	PTH	Tickets	Total
# alerts	118	120	11	249
Total	125	125	11	261
TP rate	0.94	0.96	1	0.95

Figure 5: True positive rates over the test set for the techniques L1-L3, and the true positive rate across all those techniques.

7.3 Testing

Let us now summarise the results that our detection algorithm yielded on the test set in terms of the true positive rate. Our detector alerted on 249 out of the 261 instances of Mimikatz in the test set. That corresponds to a recall number $249/261 = 95\%$, i.e. an excellent 5% false negative rate. The detailed results that we obtained by running our algorithm on the test set are displayed in Figure 5.

Experiment 4. *In order to test the false positive rate of our detection algorithm, we ran that algorithm on a total of 80 million read events that we collected over 2 days from 1.5 million MDATP customers' machines world-wide and we observed the alerts that it produced. Over the course of these two days, our detection algorithm only alerted on 230 benign read events, which yields an excellent false positive rate of $2.86 \cdot 10^{-6}$. This is an acceptable FP rate for production detectors within the MDATP product. Benign processes, such as MsMpEng, can be identified as such based on other techniques including some based on their file name and file hash.*

8 Detecting Malicious Read Access to LSASS

So far, we have built a detection algorithm that models read behaviours as linear regressions. The width of the detection interval is set to achieve satisfactory recall and false positive rate. This detector is running in production in the MDATP code base and helps to protect thousands of customers.

Next, we provide further evidence of the effectiveness of our detector by discussing interesting instances of true positives that it found, including two instances that other detection tools used at present cannot identify. Further information about those processes is displayed in Appendix D.

Figure 12 depicts a renamed instance of Mimikatz. An attacker performs such renaming in the hope that it obfuscates the presence of this malware and so avoids detection. We can see in the metadata that the flags identifying Mimikatz have not been erased or modified. Therefore, static methods that anti-viruses use or methods based on Yara rules are able to identify that this tool is malicious. The `sha1` hash of the calling process is also identifiable as being for Mimikatz – making for example use of the online tool VirusTotal.

Figure 13 describes a custom process named `lolz.exe` that we suspect is a bespoke and recompiled version of Mimikatz. We can see that all metadata related to Mimikatz have been

replaced with a suspicious “Microsoft Windows” flag. As this executable file has been customised and recompiled, Yara rules and anti-virus software would not be able to detect the running of this process.

Finally, Figure 14 shows an instance of Mimikatz being invoked remotely via PowerShell. As the calling process is the legitimate Microsoft tool PowerShell, this process will not be identified as being malicious – neither by Yara rules nor by anti-virus software. This state of affairs would be similar to a situation in which an attacker applies process injection: injecting the code of Mimikatz into another benign process P means that Mimikatz can hide within process P as static rules of anti-virus software or Yara rules cannot identify that process P contains injected code.

In both situations, the advantage of our approach is that it is independent of the calling process and its executable file. Our approach only focuses on the *behaviour* of that process, more particularly on the way that this process reads memory from the LSASS process, which holds very sensitive information.

9 Handling Windows Updates

The analyses performed in the previous sections were based on the data that MDATP collected from their customers from January to July 2019. On July 9th 2019, a major Windows update was publicly released, namely Windows 10.0.17763.615. Most MDATP customers migrated to that version in August 2019 and we noticed a slight impact of that migration on the malicious data points that we were continuously collecting. This led to the following experiment:

Experiment 5. *We collected a total of 1009 Mimikatz instances from 157 different machines over July and August 2019 in order to study the influence of this OS update. We plot in Figure 6 the malicious data corresponding to the technique L1 that steals logon passwords, collected over this period, and we colour each data point according to the version of Windows that was running. Windows versions that correspond to version 10.0.17763.615 or later are denoted as 10^+ while others are denoted as 10^- .*

In this section, we only report data from the technique for stealing logon passwords but we made very similar observations for the two other techniques. In Figure 6, we see that the data seems to be clearly separated into two different lines that correspond to credential theft actions on machines with Windows 10^+ and Windows 10^- respectively.

In order to protect current customers on Windows 10^+ against those threats, we reran our analyses on this newly collected data and built a new model for fitting the line corresponding to the new OS version, following a similar approach to that outlined in Section 7.

In order to handle future Windows updates on which the memory read behaviour of credential theft would potentially create new linear clusters, it would be of interest to develop

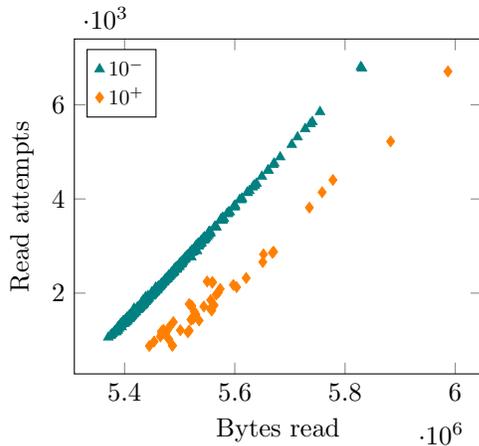


Figure 6: Influence of OS updates on memory read behaviours of logon passwords theft: 1009 password stealing behaviours collected from 157 different machines in Experiment 5 over July and August 2019. The label 10^- refers to the OS prior to the update, whereas 10^+ refers to the OS after the update.

online machine-learning algorithms that can learn how to model new linear clusters automatically as new network data is ingested. One possible idea would be to store all the malicious data points labelled with the corresponding Windows version that they were collected on. For each of the Windows versions, one would then perform a linear regression on all the corresponding data points, possibly by updating already existing regression lines. An optional optimisation step would be to merge different regression lines that would appear to be close, since memory read behaviours do not systematically change when a new OS version is released. We could also consider a multi-class classifier.

10 Minimising our Detector’s Attack Surface

In this section, we discuss how an attacker who is aware of our detection mechanism could defeat our detector, and we suggest some improvements that we could implement in order to tackle such evasions. This discussion is set within the context of *adversarial* machine learning.

Injecting random memory reads. A direct and intuitive method for an attacker to bypass our detection model is to inject random memory reads so as to fall outside of the width of the detection interval. Such a potential behaviour is depicted by the dashed arrow in Figure 7. However, we recall that we have also studied in Section 5 the behaviour of legitimate software which follow characteristic patterns. A malicious process injecting random memory reads would highly likely appear as an “isolated” point as shown in Figure 7 that we could detect using *unsupervised* detection methods.

Impersonating benign processes. More interestingly, an advanced attacker could reprogram his attack software so that its

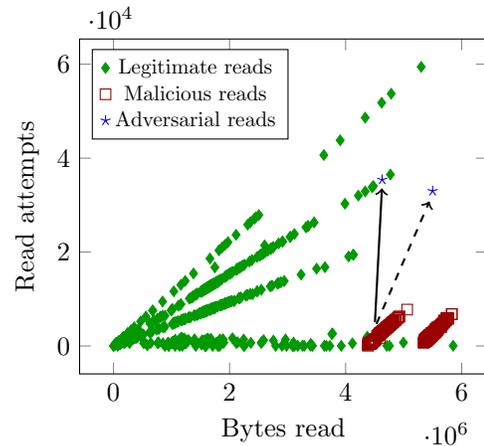


Figure 7: Adversarial memory read behaviours. The dashed arrow represents random read injections required to bypass our current model. The solid arrow represents additional read injections required to impersonate a benign software.

read behaviour meets the linear model for some non-malicious behaviour while still being able to steal credentials. Such a potential behaviour is depicted by the solid arrow in Figure 7. Our model in its current form would be unable to detect such an attacker since it would be indistinguishable from the benign software it is impersonating.

However, refining the telemetry that we get from MDATP would enable us to detect such evasions. In addition to its *volume*, we could in particular request more information about the *location* of the data that processes read.

For example, we could look at the addresses at which each read begins: we could expect legitimate processes to always start reading at some predictable – albeit randomised – set of addresses, as they always perform the same tasks.

We could also try to identify the portions of LSASS memory that are read. As legitimate processes read specific portions of memory, we could expect a malicious process to read some portions that are usually unread. Although the address space is not fixed, we could e.g. set some random points in the LSASS memory and receive flags for whether or not a process has read at this address. This would add an interesting dimension to our detector since malicious software would likely raise more flags than legitimate software, or unusual combinations of flags, since they are more likely to browse a large – and maybe contiguous – part of LSASS.

Breaking the 5min window. Our model aggregates read events over a 5-minute window. An attacker could therefore wait 5 minutes in the middle of an attack in order to bypass our detection mechanism. This would split the corresponding data point into several points that would fall outside the detection interval. In order to remedy this, it would be interesting to study the aggregation of events over an adjustable period of time, e.g. by aggregating sessions by process.

Anticipating OS updates. In the online machine learning solution described in Section 9, when a new OS version is released, defensive tools would need a source of new labelled data points in order to re-calibrate the detector as depicted in Figure 6. During this learning phase, an attacker could steal credentials without being detected, provided there is not enough data for this new OS version yet. A possible solution for this problem would be to use a set of training machines, say virtual machines set up with that new OS version, in order to run a couple of credential theft techniques on them and gather the data required to train our model.

11 Future Work

This work focused on detecting credential theft from LSASS memory. We introduced a novel method for characterising memory read behaviour and demonstrated that we can indeed build effective detectors based on this model. A natural and interesting question to pursue is whether or not we can generalise our method. In particular, we would like to study if we could adapt our model to study other parts of the memory of a Windows machine. We would also like to understand whether or not our method could generalise to other operating systems. For example, we could study system files on Linux that play a similar role in Linux to that of LSASS on Windows.

Another avenue would be to study the memory read behaviour of processes accessing a particular area in memory, in particular the memory of web browsers. This memory often contains a wide variety of sensitive information stored in different ways. This would be a particularly fascinating line of inquiry since a detection mechanism that would target a particular browser may potentially be independent of the used operating system, opening up the possibility of building cross-platform detectors. In that context, it would be of interest to study the attacker tool Lazagne, which is designed to steal passwords from web browsers.

12 Discussion

One key contribution of the work reported here is to introduce a model for analysing memory reads and to highlight two important features of such a model that enable us to characterise read sessions precisely. This precision then allowed us to build an effective detector that was not in production already.

These two features have been selected following a detailed analysis of security experts, as explained in Section 4. Those analyses were guided by the thorough examination of credential theft tools and analytical expertise of the Windows security architecture. Experimental data then naturally suggested linear models for classification. In contrast, modelling and selecting features for characterising read sessions using general machine learning techniques might not have given such successful detection results. In particular, this work demon-

strates that the application of machine learning techniques can benefit from input given by specific domain experts.

In addition, MDATP has access to a large variety of telemetry from its customers and has therefore the potential to consider read events in a much wider context where read behaviours would depend on a richer dataset such as historical data or network data. In such a case, one could use more advanced machine-learning techniques such as k -means clustering or convolutional neural networks in order to build a model for read sessions, extract sensible features, and train a classification algorithm.

Finally, we discussed in Section 10 some refinements that could improve our method. However, when building a robust detector, we believe that another important aspect to consider is to combine low level signals coming from different sources to produce more reliable – and more meaningful – alerts.

13 Conclusion

In the assumed breach scenario where an attacker has already infiltrated a system, credential theft is a common attack behaviour that allows an attacker to successfully prepare lateral network movement or privilege escalation in preparation of more advanced attacks. A particularly sensitive process targeted by credential thieves on Windows is LSASS, a process that enforces the Windows security policy. Credential theft from LSASS is currently detected based on static methods that recognise the presence of particular binary executable files that are known to perform such actions.

In this work, we introduce a model of memory read behaviour that enables us to study the read accesses of processes to LSASS memory in order to judge whether such processes are malicious or benign. Based on that model and extensive data collected from real Windows networks, we trained and tuned a linear classifier that identifies the actual act of stealing credentials from LSASS memory, rather than detecting the presence of particular malicious executable files. This enables us to detect malicious actors that were previously undetected by currently used tools. In particular, we were able to detect attacks by custom tools or malicious programs invoked remotely by PowerShell. The detection tool that contains our classifier is now running in production and we demonstrated on real customer data that it provides excellent TP and FP rates. We also established that our classifier and therefore our detection tool can cope well with operating system upgrades. Moreover, we highlighted how applying unsupervised machine learning techniques and refining the telemetry that we receive from customers could help us to protect against attackers who would try to evade our detection mechanism.

Acknowledgements. We thank anonymous reviewers and our shepherd William Robertson for their insightful advice.

References

- [1] Benjamin Delpy. Mimikatz, 2014.
- [2] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, 13(2):222–232, 1987.
- [3] Dinei Florencio and Cormac Herley. Is everything we know about password stealing wrong? *IEEE Security & Privacy*, 10(6):63–69, 2012.
- [4] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2009.
- [5] Chun-Ying Huang, Shang-Pin Ma, and Kuan-Ta Chen. Using one-time passwords to prevent password phishing attacks. *Journal of Network and Computer Applications*, 34(4):1292–1301, 2011.
- [6] Markus Jakobsson and Steven Myers. *Phishing and countermeasures: understanding the increasing problem of electronic identity theft*. John Wiley & Sons, 2006.
- [7] Parisa Kaghazgaran and Hassan Takabi. Toward an insider threat detection framework using honey permissions. *J. Internet Serv. Inf. Secur.*, 5(3):19–36, 2015.
- [8] Abhishek Kumar. Discovering passwords in the memory. *White Paper, Paladion Networks (November 2003)*, 2004.
- [9] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In Aviel D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.
- [10] Wenke Lee, Salvatore J Stolfo, and Philip K Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. New York, 1997.
- [11] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.
- [12] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.
- [13] Mike McQuade. The untold story of NotPetya, the most devastating cyberattack in history, 2018.
- [14] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 175–191, 2016.
- [15] J Mulder. Mimikatz overview, defenses and detection, 2016.
- [16] David Patten. The evolution to fileless malware. *Retrieved from*, 2017.
- [17] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 449–464, 2014.
- [18] Hung-Min Sun, Yao-Hsin Chen, and Yue-Hsun Lin. opass: A user authentication protocol resistant to password stealing and password reuse attacks. *IEEE Transactions on Information Forensics and Security*, 7(2):651–663, 2011.
- [19] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434. ACM, 2017.
- [20] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*, pages 133–145. IEEE, 1999.
- [21] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. Effective detection of android malware based on the usage of data flow apis and machine learning. *Information and software technology*, 75:17–25, 2016.
- [22] Jing Zhang, Robin Berthier, Will Rhee, Michael Bailey, Partha Pal, Farnam Jahanian, and William H Sanders. Safeguarding academic accounts and resources with the university credential abuse auditing system. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–8. IEEE, 2012.

Appendices

A Comparing Benign and Malicious Reads

This section contains Figure 8 which compares the read behaviour of legitimate and malicious software.

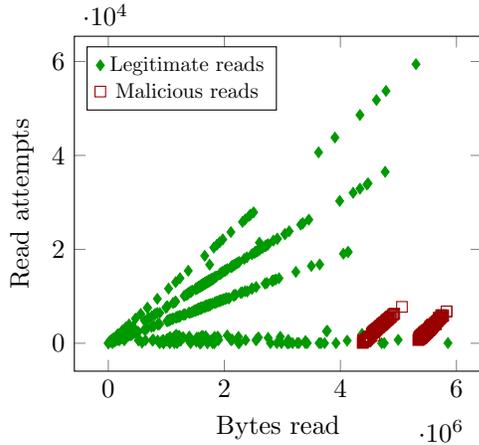


Figure 8: Comparison between benign (i.e. legitimate) and malicious memory read behaviours collected from Experiments 1 and 2.

B Modelling Malicious Read Behaviour

This section contains Figures 9 and 10 which illustrate our regression model for the memory read behaviour of Kerberos ticket thefts and pass-the-hash technique respectively.

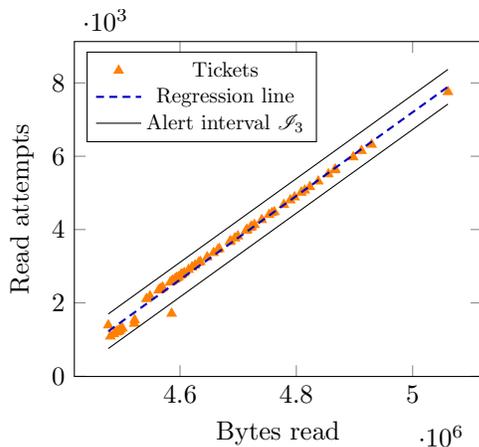


Figure 9: Memory read behaviour of Kerberos ticket thefts.

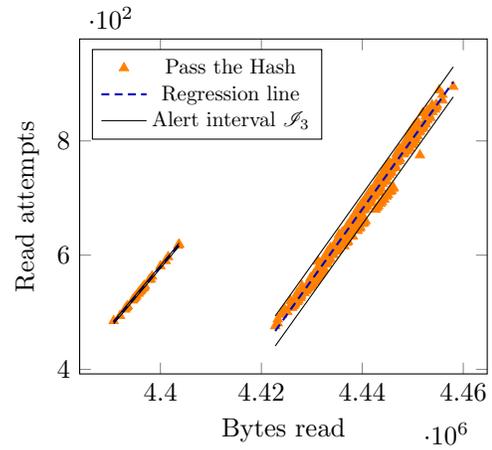


Figure 10: Memory read behaviour of passing the hash.

C Calibrating our Detector

This section contains Figure 11 which describes the influence of the length of the detection interval on the true positive and false positive rates.

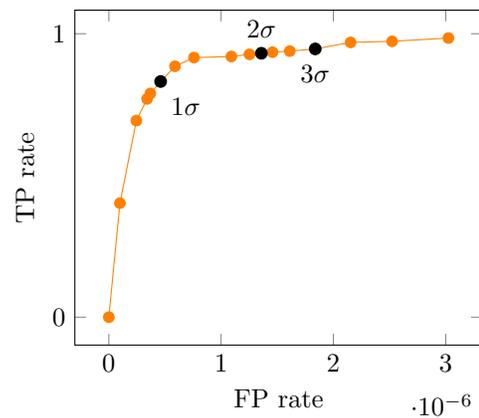


Figure 11: ROC curve: relationship between the true positive and false positive rates as the detector interval width varies.

D Detecting Malicious Memory Reads

This section contains three interesting instances of malicious memory read sessions that our mechanisms detected, including two instances that other detection tools used at present cannot identify. Figure 12 depicts a renamed instance of Mimikatz. Figure 13 describes a custom process named `lolz.exe` that we suspect is a bespoke and recompiled version of Mimikatz. Finally, Figure 14 shows an instance of Mimikatz being invoked remotely via PowerShell.

```
"FileName": "turtle.exe",
"CommandLine": "turtle.exe",
"FilePath": "E:\\",
"Sha1":
  "cb58316a65f7fe954adf864b678c694fadceb759",
"CompanyName": "gentilkiwi (Benjamin DELPY)",
"ProductName": "mimikatz",
"OriginalFileName": "mimikatz.exe",
"InternalFileName": "mimikatz",
"FileDescription": "mimikatz for Windows",
"ParentProcessName": "cmd.exe"
```

Figure 12: Credential theft tool that was renamed in order to avoid static detection but whose executable file can still be identified as malicious via its metadata.

```
"FileName": "lolz.exe",
"CommandLine": "lolz.exe",
"FilePath": "E:\\",
"Sha1":
  "fc03697be2ed32f844aa153460ef5c82f58b06a0",
"CompanyName": "Microsoft Windows",
"ProductName": "Microsoft Windows",
"OriginalFileName": "Microsoft Windows",
"InternalFileName": "Microsoft Windows",
"FileDescription": "Microsoft Windows"
"ParentProcessName": "cmd.exe"
```

Figure 13: Process performing credential theft that has been customised and recompiled so that its executable file bypasses *static* detection mechanisms. Our detector does detect this process as malicious by looking at its memory read behaviour.

```
"FileName": "powershell.exe",
"CommandLine": "\" powershell.exe\" ... { IEX
  $mimi; Invoke-Mimikatz -DumpCreds } ...",
"FilePath": "C:\\Windows\\System32\\
  WindowsPowerShell\\v1.0",
"Sha1":
  "6cbce4a295c163791b60fc23d285e6d84f28ee4c"
"CompanyName": "Microsoft Corporation",
"ProductName": "Microsoft Windows Operating
  System",
"OriginalFileName": "PowerShell.EXE",
"InternalFileName": "POWERSHELL",
"FileDescription": "Windows PowerShell"
"ParentProcessName": "ai_python.exe"
```

Figure 14: Credential theft tool invoked remotely via PowerShell. The malicious executable file is not physically on the machine, so its presence is not detected by current detectors. But our detector triggers an alert when PowerShell performs this action, as its memory read behaviour becomes malicious.

EnclavePDP: A General Framework to Verify Data Integrity in Cloud Using Intel SGX

Yun He^{1,2}, Yihua Xu³, Xiaoqi Jia^{1,2*}, Shengzhi Zhang³, Peng Liu⁴, and Shuai Chang^{1,2}

¹{CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences

²School of Cyber Security, University of Chinese Academy of Sciences

³Metropolitan College, Boston University

⁴Pennsylvania State University

Abstract

As the cloud storage service becomes pervasive, verifying the integrity of their outsourced data on cloud remotely turns out to be challenging for users. Existing Provable Data Possession (PDP) schemes mostly resort to a Third Party Auditor (TPA) to verify the integrity on behalf of users, thus reducing their communication and computation burden. However, such schemes demand fully trusted TPA, that is, placing TPA in the Trusted Computing Base (TCB), which is not always a reasonable assumption. In this paper, we propose EnclavePDP, a secure and general data integrity verification framework that relies on Intel SGX to establish the TCB for PDP schemes, thus eliminating the TPA from the TCB. EnclavePDP supports both new and existing PDP schemes by integrating core functionalities of cryptography libraries into Intel SGX. We choose 10 existing representative PDP schemes, and port them into EnclavePDP with reasonable effort. By deploying EnclavePDP in a real-world cloud storage platform and running the 10 PDP schemes respectively, we demonstrate that EnclavePDP can eliminate the dependence on TPA and introduce reasonable performance overhead.

1 Introduction

Nowadays, many organizations demand to keep their data records, and then perform deep analysis over the data using machine learning or other techniques for their business purposes. For instance, e-health companies offer optimized health care plan for customers by analyzing customers' health records. However, not all organizations are able to build and manage their private data storage platform due to the high cost of building and maintaining such a platform. Hence, cloud storage service has become quite popular, due to the features like pay-as-you-go, elasticity, cost-saving, maintenance, etc. There are many popular cloud storage services today, such as Dropbox, Google Drive, Amazon S3, One Drive, etc.

However, users will lose control of their data stored on the cloud platform, which is an inherent issue in such data outsourcing model. Although the service providers can be bound by a Service Level Agreement (SLA) to ensure the data integrity, users still cannot fully trust them. On one hand, the cloud servers are not immune to data loss or corruption even the cloud providers are faithful to protect the outsourced data. For instance, Dropbox, Amazon and Tencent Cloud lost data due to improper operations, inadvertent administration errors or system bugs [1–3]. Although these incidents occurred unintentionally, the service providers may not immediately inform the data loss incidents to users to protect their reputation (i.e., service providers are “imperfect and selfish” [4]). For example, according to [5], healthcare data breaches are identified after 84.78 days and customers are notified after additional 68.31 days on average. On the other hand, a service provider may be actively malicious: deleting data that is infrequently accessed to save storage space but still charging users for the deleted data [6], or keeping fewer replicas violating the SLA.

In recent years, numerous data integrity verification approaches [7–19] have been proposed to ensure the integrity of the outsourced data. These approaches are referred to as Provable Data Possession (PDP) schemes. Such PDP schemes provide probabilistic guarantees that the outsourced data has not been maliciously tampered with, without accessing the entire data from the cloud storage server. PDP schemes (e.g., APDP [9], etc.) usually generate metadata (or tag) using the original data, and upload the metadata together with the original data to the cloud storage servers. The proof of data integrity is generated by cloud storage servers using this metadata and verified by the data owner. Other PDP schemes [14, 17–19] were proposed to support public auditing for multiple users via a Third Party Auditor (TPA). However, the TPA may steal users' private data (honest but curious TPA) or even conduct collusion attacks with the cloud service provider (inherently malicious TPA). Besides the trustworthiness concern, deploying TPA also involves extra cost.

In this paper, we propose **EnclavePDP** (Enclave-protected Provable Data Possession), a practical and general frame-

*Corresponding author: jiaxiaoqi@iie.ac.cn

[†]Key Laboratory of Network Assessment Technology, CAS

[‡]Beijing Key Laboratory of Network Security and Protection Technology

work to verify the integrity of the outsourced data on cloud platforms relying on the Trusted Execution Environments (TEEs), i.e., Intel SGX [20], thus eliminating TPAs and reducing both the computation and communication burdens from users. We implemented a prototype of EnclavePDP using Intel SGX and ported the core functionalities of Intel SGX SSL crypto library [21], Intel SGX GMP library [22], and the PBC [23] (Pairing-Based Cryptography) library into EnclavePDP. Then, 10 representative PDP schemes, i.e., APDP [9], CPOR [12], SEPDP [10], MRDPDP [11], PPPAS [19], DHT-PA [18], SEPAP [17], DPDP [7], FlexDPDP [8], and a basic Message Authentication Code (MAC) based scheme (MAC-PDP), are implemented on EnclavePDP with reasonable effort. We evaluated EnclavePDP on a real-world cloud storage service, FastDFS [24]. Experimental results show that EnclavePDP introduced negligible overhead to the response time per PDP request for all the 10 PDP schemes on different file sizes (1GB and 16KB), varying from 1.0% to 24.5%.

We summarize the main contribution of the paper as below:

- We proposed and implemented EnclavePDP, a novel and generic framework that can securely verify the integrity of the outsourced data relying on Intel SGX, thus eliminating the dependency on TPAs. The core functionalities of various cryptographic libraries are tailored and ported into Intel SGX to support both the existing and new PDP schemes, and 10 representative PDP schemes are implemented in EnclavePDP with reasonable effort.
- We performed a comprehensive evaluation of EnclavePDP by deploying it on a real cloud storage service, FastDFS. All the 10 PDP schemes are evaluated in EnclavePDP in the aspects of code base in Intel SGX and overhead of response time, thus eliminating the performance concerns of running PDP schemes in Intel SGX.

2 Background

2.1 Provable Data Possession in Clouds

To verify the integrity of the outsourced data on cloud platforms, lots of PDP schemes [7–16] were proposed. Generally, PDP schemes consist of two phases: a setup phase and a verification phase. In the setup phase, the client (or the data owner) generates keys (private or public, depending on the scheme), as well as metadata (or tag) using the keys and the original data. The metadata (or tag) and the original data are uploaded to the cloud storage server. In the verification phase, the client constructs a challenge that contains a random subset of file blocks, and sends the challenge to the prover (i.e., the cloud storage server). The prover uses the challenge, the metadata (or tag) and the file blocks to compute a proof of data possession and then sends it back to the client. The client uses the proof to verify that the data on the cloud is still intact. In addition, many literature surveys, e.g., [25–29] made a comprehensive comparison among existing PDP schemes.

Below we choose four aspects: types of data, retrievability, encryption and auditing, to discuss the existing PDP schemes.

Types of data: There exist two types of data: static data and dynamic data. Static data (e.g., data archive, backups) is never modified but appended only, whereas dynamic data is frequently changed due to operations like update, write and delete. Some PDP schemes, e.g., APDP [9], are only suitable for static data, because they need re-generate tags of the complete file whenever new data is inserted. In contrast, other schemes like SEPDP [10] support dynamic data operations.

Retrievability: Generally, PDP schemes only provide probabilistic guarantees of the data integrity, i.e., identifying data corruption without data recovery, e.g., [8–11, 30], etc. In contrast, POR (Proof of Retrievability) schemes provide the guarantee that the data is intact and still retrievable even after corrupted by using the redundant encoding, e.g., CPOR [12], Mirror [16], Iris [31], etc.

Encryption: Some PDP schemes utilize symmetric key encryption to achieve scalability/efficiency, e.g., SEPDP [10] uses symmetric key encryption and cryptographic hash functions, while others use asymmetric key encryption for better security, e.g., APDP [9] uses RSA-based homomorphic verifiable tags (HVT) as the metadata.

Auditing: PDP schemes either support private auditing or public auditing. For the former, the verifier is always the data owner, e.g., APDP [9], SEPDP [10], FlexDPDP [8], etc. For the latter, the TPA sends challenges and verifies proofs on behalf of the data owner to reduce the computation and communication overhead of the data owner. Public auditing schemes can be further categorized into privacy preserving (e.g., PPPAS [19], DHT-PA [18]) and non-privacy preserving (e.g., PoS [32], SEPAP [17], MHT-PA [6]) schemes. It is worth noting most of the public auditing schemes are implemented using the BLS [33] signature cryptographic primitive.

2.2 Intel SGX

Intel SGX [20] creates an isolated code execution environment, which enables applications to maintain data confidentiality and integrity. Even the privileged software (OS, hypervisor and BIOS) cannot violate the protection of Intel SGX. Note that we do not consider side-channel attacks against SGX, which can be addressed orthogonally by corresponding countermeasures (e.g., [34]).

Enclave. Intel SGX constructs trusted execution environments referred to as *enclaves* and creates an encrypted memory region called Enclave Page Cache (EPC) for enclaves to store code and data. SGX uses a hardware Memory Encryption Engine (MEE) [35] to encrypt/decrypt the enclave data, and also provides a hardware access control mechanism to prevent illegal access to the enclave memory. An Intel SGX application generally contains two parts: secure code (enclave) and non-secure code (non-enclave or application). The application needs to launch the enclave, and uses

ecall/ocall interfaces to switch control between the enclave and the non-enclave. Since privileged operations (e.g., system calls) cannot be executed inside enclaves, ocall is invoked to execute those privileged operations indirectly.

SGX Remote Attestation. Intel SGX remote attestation [36] is to ensure that the enclave is correctly initialized on a remote SGX enabled platform. It evaluates the enclave identity, its structure, the integrity of the code inside the enclave. Furthermore, remote attestation can provide shared secret between the enclave application and its owner to setup a secure communication channel over the untrusted network.

Sealing. Enclaves can write confidential data to persistent storage securely using *sealing* [36], a mechanism to encrypt and authenticate the enclave data. Each enclave is provided with a sealing key, derived from an enclave identity (either Enclave Identity or Signing Identity), private to the executing platform and the enclave. Data sealed against Enclave Identity (MRENCLAVE) can only be decrypted by the same enclave, whereas data sealed against Signing Identify (MRSIGNER) can be unsealed by any enclave signed by the same developer.

3 Overview

3.1 System and Threat Model

We consider a cloud storage scenario where usually three primary entities exist: *clients or users*, *Cloud Storage Service (CSS)*, and *TPA*. Specially, clients have a large amount of data to be stored on the cloud, and CSS is managed by the Cloud Service Provider (CSP) to provide data storage service (typically with a large amount of storage space and computational resources). To save the computational resources as well as the online burden potentially incurred by the periodic data integrity verification, clients resort to TPA (with extra capabilities that clients do not have, e.g., keeping always online) to verify the integrity of their outsourced data on cloud.

We assume the threats to the integrity of users' outsourced data on cloud can be both internal and external on the cloud storage platform, e.g., software bugs, hardware failures, malicious or accidental management errors, revenue-motivated hackers, etc. Moreover, the cloud storage platform may intend to hide the data corruption incidents from users to maintain its reputation. Most prior works, e.g., MHT-PA [6], SEPAP [17], usually rely on TPA to provide a cost-effective way for users to verify the integrity of their outsourced data, with the assumption that TPA is reliable and trustworthy. However, such assumption is not always valid, since TPA could be (1) honest but curious, learning the users' data after the audit as described in PPPAS [19], and (2) untrusted, conducting collusion attack with the untrusted cloud service providers. Hence, the proposed solution in this paper does not rely on TPA. We assume the remote CPU (with Intel SGX security features) running on the cloud storage platform is trusted. We also assume that the adversary cannot extract secrets within

the CPU packages, which implies that we trust CPUs to protect code and data hosted inside TEEs. Side-channel and denial-of-service attacks are outside the scope of this paper.

3.2 Motivation of Using Intel SGX

In this paper, we mainly focus on those PDP schemes that rely on TPA to verify the integrity of the outsourced data, since users' computation resources as well as online burden can be significantly reduced by TPA. However, such PDP schemes are still limited in the following aspects. **(P1) The honest but curious TPA.** Generally, TPA needs to be fully trustworthy, exactly following the PDP schemes to execute the core verification functionality. However, an honest but curious TPA may potentially learn users' data through the procedures of challenging and verifying [19] it gets involved in. **(P2) Collusion attack with untrusted cloud providers.** Although a few privacy-preserving public auditing schemes, e.g., PPPAS [19] and DHT-PA [18], can be used to address the data breach issues, they are still limited in eliminating the collusion attack when the TPA collaborates with the cloud storage server to deceive users. **(P3) Communication overhead.** The communication overhead (sending challenges and proofs between the cloud storage server and the verifier running in the TPA) is not negligible.

Fortunately, Intel SGX provides the trusted execution environment, enclave, which can be leveraged to solve the above problems. First, Intel SGX prevents the underlying untrusted OS or hypervisor from accessing the code/data inside the enclave. Hence, the PDP schemes can run faithfully in enclave on untrusted platforms (i.e., the cloud storage servers), thus eliminating the dependency on TPA (solving P1 and P2). Second, the enclave can also protect the private keys used by the PDP schemes against leaking to untrusted components, and it can also protect the integrity of verification against malicious modification. Therefore, we can also deploy private auditing PDP schemes inside the enclave, which reduces the computation overhead of the data owners and provides public-auditing-like support. Finally, EnclavePDP can be deployed on any of the cloud servers (as long as the underlying Intel CPU supports SGX), thus it can be co-located with the cloud storage services on the same physical machine. Hence, the communication overhead between the verifier in EnclavePDP and the cloud storage services (inter-process communication) is negligible (solving P3), compared with that of the native PDP schemes (network communication).

3.3 Possible Concerns of Using Intel SGX

Compatibility. The implementation of PDP schemes depends on some cryptography libraries (e.g., OpenSSL, PBC [23], etc.). Therefore, these libraries must be ported into Intel SGX before implementing PDP schemes inside the enclave. Currently, two libraries have been ported into enclave: (i)

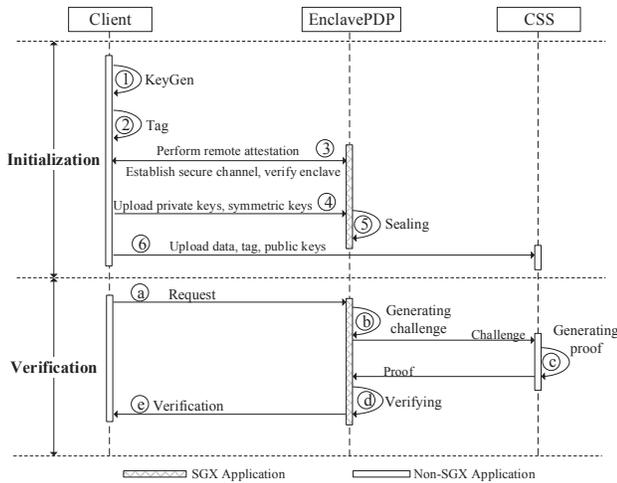


Figure 2: The workflow of EnclavePDP

EnclavePDP Initialization. In the initialization phase, clients (data owners) cooperate with EnclavePDP to complete the necessary setup:

(1) *Key Generation*: Several keys will be generated by the client, i.e., a pair of public key (pub_k) and private key (pri_k), as well as a symmetric key (sk). The public key is available to the cloud storage services and will be used to generate proofs, while the private key, owned by the client and EnclavePDP, is used to generate tags as well as challenges, and verify proofs. EnclavePDP uses the symmetric key to encrypt the verification result and sends it back to the client.

(2) *Data Tagging*: The client generates tags for the original data using the private key. For some PDP schemes (e.g., DHT-PA [18]), the client also generates additional data structures (e.g., Dynamic Hash Table) to record extra information, e.g., data version, timestamp, etc., to support dynamic auditing.

(3) *Remote Attestation*: The client can upload the EnclavePDP executable to the cloud server running on Intel SGX, and start the EnclavePDP remotely. After starting EnclavePDP, the client attests it using Intel SGX remote attestation to verify the code integrity of EnclavePDP, and subsequently performs ECDH [38] protocol to create a secure communication channel for the following operations.

(4) *Secrets Uploading*: The private key and the symmetric key will be sent to EnclavePDP via the secure channel established in Step (3). Other security-sensitive data (e.g., dynamic hash table of DHT-PA schemes [18]) used to support dynamic auditing will also be uploaded to EnclavePDP.

(5) *Sealing*: When EnclavePDP receives the private key and the symmetric key, it encrypts them using Intel SGX Sealing technique and stores the sealed (i.e., encrypted) keys on disk. Other security-sensitive data, (e.g., dynamic hash table of DHT-PA [18]), should be sealed in the same way.

(6) *Data Uploading*: The client uploads the public key, the data and all the tags to the cloud storage services (CSS).

EnclavePDP Verification. In the verification phase, the client issues a PDP request (e.g., via HTTP) to the cloud storage service, which forwards the request (e.g., via TCP

sockets) to the *Request handler* (step ① in Figure 1). Note that the client can also directly issue PDP requests to EnclavePDP. The request then will be forwarded to the *Challenger* via `ecalls` interface, and the *Challenger* generates a challenge (step ② in Figure 1) for the file indicated in the request (typically via file name or file path). The *Request handler* will transmit the challenge to the *Prover*, a regular application (i.e., non-SGX application) that can access the data stored in the cloud storage servers. Note that the cloud storage services can entitle the *Prover* to access data directly or provide APIs for the *Prover* to access data indirectly. The *Prover* reads the data from the cloud storage servers, and uses such data (i.e., file blocks, tags) and the challenge to generate a proof of data possession. The proof will be sent back to the *Request handler* (step ③ in Figure 1) and then forwarded to the *Verifier* (step ④ in Figure 1). The *Verifier* uses the proof to verify whether the cloud storage servers actually possess the correct data, and returns a verification response encrypted using the symmetric key sk to the *Request handler*. Finally, the client will receive (step ⑤ in Figure 1) and decrypt the verification response using the same sk to ensure the confidentiality and integrity of it. Note that the *Request handler* is also designed to be able to generate verification requests periodically on behalf of the client, and forward these requests to the *Challenger*, followed by other steps mentioned as above. Finally, EnclavePDP will create a tamper-free (encrypted by the enclave) verification log that will be forwarded to the client if necessary.

4.3 Key Management

4.3.1 Private Key Protecting

The private key used by the PDP schemes can never be exposed to the cloud storage services. In the initialization phase, the client establishes a secure channel to upload the private key and EnclavePDP encrypts the private key using SGX Sealing technique. The confidentiality and integrity of the keys are guaranteed by two conditions: (i) the ECDH protocol is executed in the enclave, which guarantees the confidentiality and integrity of the ECDH computation; (ii) The sealed private key is bound to a signing authority (developer), so only the enclave signed by the same authority can unseal it.

4.3.2 Private Key Loading

Before performing challenge or verification operations, the *request handler* (running outside of the enclave) firstly reads the sealed private key from the disk and invokes `ecalls` provided by *Private Key Loader* to unseal the private key inside the enclave. Note that the symmetric key sk is also loaded and unsealed in the enclave. When generating challenge (or verification), the *Challenger* (or the *Verifier*) uses the unsealed private key to generate challenges (or verify the proof). To reduce enclave transitions caused by `ecalls`, the private keys used recently are stored in the private key buffer. Thus, the

Table 1: A brief comparison of the ten PDP schemes

	Dynamic	Retrievability	Public	Encryption
MACPDP	X	PDP	X	Sym.
APDP [9]	X	PDP	X	Asym.
MRPDP [11]	X	PDP	X	Asym.
SEDP [10]	X	PDP	X	Sym.
CPOR [12]	X	POR	X	Sym.
DPDP [40]	✓	PDP	X	Asym.
FlexDPDP [8]	✓	PDP	X	Asym.
PPAS [19]	X	PDP	✓	Asym.
SEPAP [17]	✓	PDP	✓	Asym.
DHT-PA [18]	✓	PDP	✓	Asym.

* Note: "✓" means "support"; "X" means "not support"; "Sym." means symmetric encryption; "Asym." means asymmetric encryption.

Private Key Loader will firstly check if the required private key already exists in the private key buffer. If so, it returns the private key directly. Otherwise, it will demand the *request handler* to load the sealed private key from the disk and unseal the private key into the private key buffer. To save the enclave memory, the LRU (Least Recently Used) strategy is utilized to refresh the private key buffer.

5 Implementation

We implemented a prototype of EnclavePDP on a Linux platform, based on Intel SGX SDK 2.4, Intel SGX Driver 1.0, Intel SGX SSL library integrated with OpenSSL 1.1.0i, Intel SGX GMP library and an enclave-supported PBC library trimmed based on pbc-0.5.14. For generality and scalability, the *Request handler* utilizes Linux epoll [39] mechanism to provide support for multi-thread execution and concurrent responses. The requests to verify data integrity are encapsulated into TCP sockets and forwarded to EnclavePDP, which eases the deployment of EnclavePDP on third party cloud services.

5.1 Porting PDP Schemes

PDP Implementation. We chose 10 representative PDP schemes, which cover the taxonomy described in Section 2.1 as in Table 1. Most of the PDP schemes can be implemented in Intel SGX quite straightforward, but the following issues need to be addressed for other PDP schemes.

(1) For MAC-PDP, to avoid frequent I/O operations from the enclave and reduce the EPC memory consumption. the MAC of the file blocks to be verified is not re-computed inside the enclave. Instead, the prover (running on the cloud storage server) re-computes the MAC and also loads the encrypted tags (i.e., MACs of file blocks encrypted by the private key and uploaded to the cloud storage server during the initialization phase) associated with these file blocks into non-EPC memory. Then the verifier (inside the EnclavePDP) decrypts the tags to get the original MAC and compares it with the MAC computed by the prover.

(2) Some PDP schemes (e.g., SEPAP and DHT-PA) design an extra data structure to record the data property information (e.g., timestamp, version) used to perform dynamic auditing

³ Such additional data structures should be uploaded to the TPA (when involved), protecting their integrity from the cloud storage server. In contrast, EnclavePDP encrypts these data structures using the private key and upload them to the remote cloud server during the initialization phase. In the verification phase, EnclavePDP decrypts them in the enclave and uses them to verify the proofs.

(3) During the initialization phase, DPDP generates root metadata based on the Rank-Based Authenticated Skiplist (RBASL) to verify its integrity, while FlexDPDP generates root metadata based on FlexList to verify its integrity. Similar as (2), EnclavePDP also encrypts the root metadata and uploads it to the cloud storage server, and then decrypts it inside the enclave in the verification phase.

Trimming Intel SGX SSL library. Intel SGX SSL [21] is to provide cryptographic service for enclave applications based on OpenSSL library. It includes lots of functionalities that are unnecessary to implement PDP schemes, e.g., *des*, *rc2* and *md4*, etc. To save the enclave memory consumption, we trimmed the native implementation of SGX SSL by removing those unnecessary modules from the configuration file at the compilation time. Finally the size of the trimmed SGX SSL library decreases 26.1% (from 4.6MB to 3.4MB).

Porting PBC library. Public auditing schemes (e.g., PPAS [19], SEPAP [17] and DHT-PA [18]) are all based on the BLS signature cryptographic primitive implemented in the PBC library [23], which is not supported by Intel SGX yet. Therefore, we ported the PBC library into SGX to make it easy to port other existing or develop new BLS-based schemes in EnclavePDP. We only ported those functions required by the public auditing schemes into SGX to provide a lightweight PBC library, thus reducing the memory consumption of EnclavePDP. Note that some of those functions need a bit tuning. For instance, generating random numbers is a quite frequent operation for most PDP schemes, the PBC library generates random numbers using the */dev/urandom* pseudo file on Linux platform. However, code running in the enclave cannot perform I/O operations directly. Hence, we use Intel RDRAND instruction [41] when porting the random number generation function in the PBC library.

5.2 Protecting Enclave Binary Integrity

The implementation of the PDP schemes inside the enclave is essentially an executable binary running on the untrusted cloud platform. Hence, the adversaries may reverse-engineer the binary enclave shared object to extract the code logic. We utilized Intel SGX PCL technique [42] to encrypt the enclave shared object (.so) at build time and decrypt it at enclave load time. Moreover, the untrusted cloud providers may create a fake enclave to perform ECDH [38] protocol with the data

³In particular, when generating tags for the original data, SEPAP will create a doubly linked info table (DLIT), while DHT-PA scheme will create a dynamic hash table (DHT).

owner to steal the private keys. To defeat such threat, the data owner periodically requests enclave to return its enclave measurement (constructed by invoking the *EREPORT* instruction, which can only be executed inside the enclave), and compares it with local backup measurement. The successive operations can only be continued upon a match of the measurements. Note that malicious cloud providers may create a copy of EnclavePDP and execute this copy, but they cannot reveal any secret data inside the enclave. The copy of EnclavePDP may cause DoS attack, which is out of scope of this work.

5.3 Integration with Cloud Storage Service

In order to deploy EnclavePDP on existing cloud storage services easily, we exposed high-level interfaces (e.g., TCP sockets) for users or cloud storage services to submit/return PDP requests/responses.⁴

We deployed the prototype of EnclavePDP on FastDFS [24], an open source high performance distributed file system (DFS). FastDFS has two major functionalities: tracker and storage. The former conducts scheduling and load balancing for file access. The latter performs file management including: file storing, file syncing, providing file access interface. We extended the *fastdfs-nginx-module* of FastDFS for user to easily submit integrity verification requests, e.g., issuing a *http get* request. When receiving requests submitted by users, the *fastdfs-nginx-module* forwards the requests to EnclavePDP (runs as a daemon on the storage servers) and waits for the verification result returned by EnclavePDP. The implementation of integrating EnclavePDP with FastDFS is less than *300 lines of C code*. Note that the *Prover* runs on the storage server of FastDFS, so it can access the outsourced data directly and generate proofs on behalf of FastDFS. As for the closed source cloud storage services (e.g., Amazon S3), EnclavePDP can only invoke the public APIs exposed by those cloud storage services to access the outsourced data. Current implementation of EnclavePDP supports the integrity check of the data stored on Amazon S3 using AWS C++ SDK, with around *70 lines of C++ code* added into EnclavePDP and without any changes to Amazon S3 platform. However, EnclavePDP needs to download all the data to local disk and performs verification, because Amazon S3 does not support random access to different data blocks.

⁴The cloud storage service needs to: (1) allow users to submit PDP requests and forward the PDP requests to EnclavePDP; (2) allow the *Prover* process to access the outsourced data directly, or provide APIs for the *Prover* to access the data indirectly. Recall that the *Prover* is a non-SGX application designed to generate proofs on behalf of the cloud storage services, which makes it possible to integrate EnclavePDP with existing cloud storage services with as few changes as possible.

6 Evaluation

6.1 Experimental Setup

We deployed EnclavePDP and FastDFS on Microsoft Azure Confidential Computing (ACC) [43] VMs supporting Intel SGX. Each VM runs Ubuntu 16.04.1 LTS with kernel version 4.15.0-1036 on a platform with an Intel(R) Xeon(R) E-2176G CPU (4 cores, 3.70 GHz, and 12 MB cache) and 16 GB RAM. We ran FastDFS v5.12 on four VMs, one VM as the tracker server and the others as storage servers. The tracker server takes charge of scheduling and load balancing for file access, and is also extended to dispatch PDP requests to other storage servers. In particular, FastDFS utilizes its Nginx module (i.e., *fastdfs-nginx-module* that is built on *nginx-1.15.4*) to interact with the user, thus we extend this module to handle the PDP requests submitted by the user. EnclavePDP runs as a daemon on the storage servers. When the tracker server receives PDP request, it dispatches the PDP request to the EnclavePDP running on the corresponding storage servers. To evaluate the throughput of EnclavePDP when handling concurrent requests, we used a popular workload testing tool, Apache JMeter, to simultaneously issue integrity verification requests to EnclavePDP at different speed (requests/second). Apache JMeter runs on a local computer with Ubuntu 16.04.1 LTS equipped with Intel(R) Core(TM) i7-7700HQ CPU.

6.2 Analysis of TCB

We measured the change of the TCB code base after porting the *Challenge* and *Verify* operations into Intel SGX, as shown in Table 2. We only focus on the core part of the implementation of those PDP schemes when measuring the SLOC (Source line of code), and ignore other code like I/O operations, sockets, etc. All the PDP schemes include *Challenge* and *Verify* operations, which are two security-sensitive functions. To guarantee the confidentiality of private keys used to generate challenges or verify proofs, loading private keys into enclave is the third security-sensitive function. For DPDP and FlexDPDP, there is an extra verification against the integrity of the Rank-based Authenticated SkipList and FlexList respectively. Therefore, there exists the fourth security-sensitive function for those two schemes. Accordingly, each security-sensitive function is associated with an *ecall* interface. Hence, each PDP request will conduct three or four *ecall* crossings (i.e., traps into enclave) depending on the specific schemes.

As in Table 2, the security-sensitive SLOC of native PDP varies from 7% to 33%, while the security-sensitive SLOC after porting them into enclave varies from 8% to 36%. Take APDP as an example. Its native implementation totally contains 1348 SLOC, among which 300 SLOC is security-sensitive (account for 22% of the total). After porting it into enclave, the security-sensitive SLOC increases to 350 SLOC

Table 2: TCB size of EnclavePDP

Schemes	SLOC	Security -sensitive SLOC	Security -sensitive functions	SGX -enabled SLOC
MACPDP	1483	115 (7%)	3	121 (8%)
APDP [9]	1348	300 (22%)	3	350 (25%)
MRPDP [11]	1440	476 (33%)	3	624 (43%)
SEDPDP [10]	1259	106 (8%)	3	153 (12%)
CPOR [12]	1057	167 (15%)	3	210 (19%)
DPDP [7]	950	117 (12%)	4	145 (15%)
FlexDPDP [8]	945	139 (14%)	4	158 (16%)
PPPAS [19]	1012	199 (20%)	3	249 (24%)
SEPAP [17]	620	162 (26%)	3	225 (36%)
DHT-PA [18]	720	187 (26%)	3	255 (35%)

(25% of the total). Such increase mainly results from extra functionalities, such as private key loading, challenges backup/destroy, decrypting other security-sensitive data (e.g., doubly linked info table, dynamic hash table), etc. Additionally, we also quantitatively measured the SLOC of those three enclave-supported libraries: Intel SGX SSL library contains about 138.4K SLOC; Intel SGX GMP contains 163.4K SLOC; PBC library contains 29.9K SLOC.

6.3 Evaluation of Challenger and Verifier

Given the amount of data outsourced on the cloud, it is inadvisable to challenge all data blocks at once to verify the integrity. Instead, the sampling verification is used by most PDP schemes, that is, to achieve high-accuracy verification by only checking a portion of the data at once. In particular, [9, 11, 18, 19] demonstrated that if t fraction of data is corrupted, randomly sampling c blocks will detect such corruption with the probability $P = 1 - (1 - t)^c$. When $t = 1\%$, the verifier only needs to verify 460 randomly chosen blocks to detect such corruption with the probability larger than 99%. Hence, in all the following experiments, we choose 460 as the maximum number of challenge blocks⁵, even for large files with much more file blocks. When the number of the total file blocks is less than 460, the verifier challenges all the file blocks instead. We measured the performance of performing the *Challenge* and *Verify* operations inside the enclave, and compared it with the native implementation below. Note that the time involved in sending challenges/proofs and reading data is ignored for both of the two cases.

6.3.1 Overhead of Challenge Operation

Figure 3 depicts the time (in μs) of generating challenges for both enclave-enabled and native implementation with varying file sizes. For all the 10 PDP schemes, both enclave and native implementation demonstrate similar changes over different file sizes. The challenge operation time of APDP, MRPDP and SEDPDP is relatively constant regardless of file size, because

⁵The block size is 16KB for APDP, and 4KB for other PDP schemes.

their challenge operations just produce a random seed used to generate the random block set to be verified, which is independent of the file size. For the other seven PDP schemes, as the file size increases, the challenge operation time first increases and then becomes constant. This is because those schemes generate a random n -element set for the challenge, whose size increases as the file size increases. It reaches a constant (i.e., the maximum number of challenge blocks) when the number of file blocks exceeds the maximum number of challenge blocks (460 as described above).

Comparing with the native PDP schemes, APDP and MRPDP saw an increase of 18.2% and 18.1% of the challenge operation time respectively. MAC-PDP, DPDP, FlexDPDP and CPOR incurred 62%, 50%, 41.5% and 180% overhead when their challenge operation time reaches a constant. The three BLS-based schemes, i.e., PPPAS, DHT-PA, SEPAP, imposed similar overhead, 89.7%, 85.3% and 84.3% respectively. The challenge operation time of SEDPDP increased nearly 1.9 times. Actually the difference of overhead results from the challenge operation time of each PDP scheme. For instance, the challenge operation for native SEDPDP is below 4 μs for varying file sizes, which magnifies the impact of `ecall` overhead, thus causing nearly 1.9 times overhead. In contrast, the challenge operation for native APDP is from 250 μs to 300 μs for varying file sizes, thus causing merely 18.2% overhead.

Observation 1. The overhead of the challenge operation is not proportional to the security-sensitive SLOC. PDP schemes in the same category introduce similar overhead. Enclave-enabled challenge operation time is still in the scale of microsecond (μs), which should have little impact on practical applications.

6.3.2 Overhead of Verify Operation

Figure 4 depicts the time of executing the verify operation for both enclave-enabled and native implementation with varying file sizes. The verify operation time of native PDP schemes varies significantly. In particular, the verify operation time of SEDPDP, MAC-PDP and CPOR is in the scale of microsecond (μs), but in the scale of millisecond (ms) for APDP, MRPDP, DPDP and FlexDPDP (FDPDP). For the other three BLS-based schemes (PPPAS, DHT-PA and SEPAP), their verify operation time is in the scale of second (s).

Observation 2. RSA-based schemes (ms) are an order of magnitude slower than symmetric-based schemes (μs), because the RSA-based modular exponential operations are complicated and expensive. BLS-based schemes (s) is another order of magnitude slower, probably due to the inherent drawback of the complicated and slow computation of BLS signatures (e.g., curves pairing) [44].

Regarding enclave-enabled implementation of PDP schemes, executing the verify operation inside the enclave imposed 17.1%, 12.7% and 24.7% overhead for APDP, MRPDP

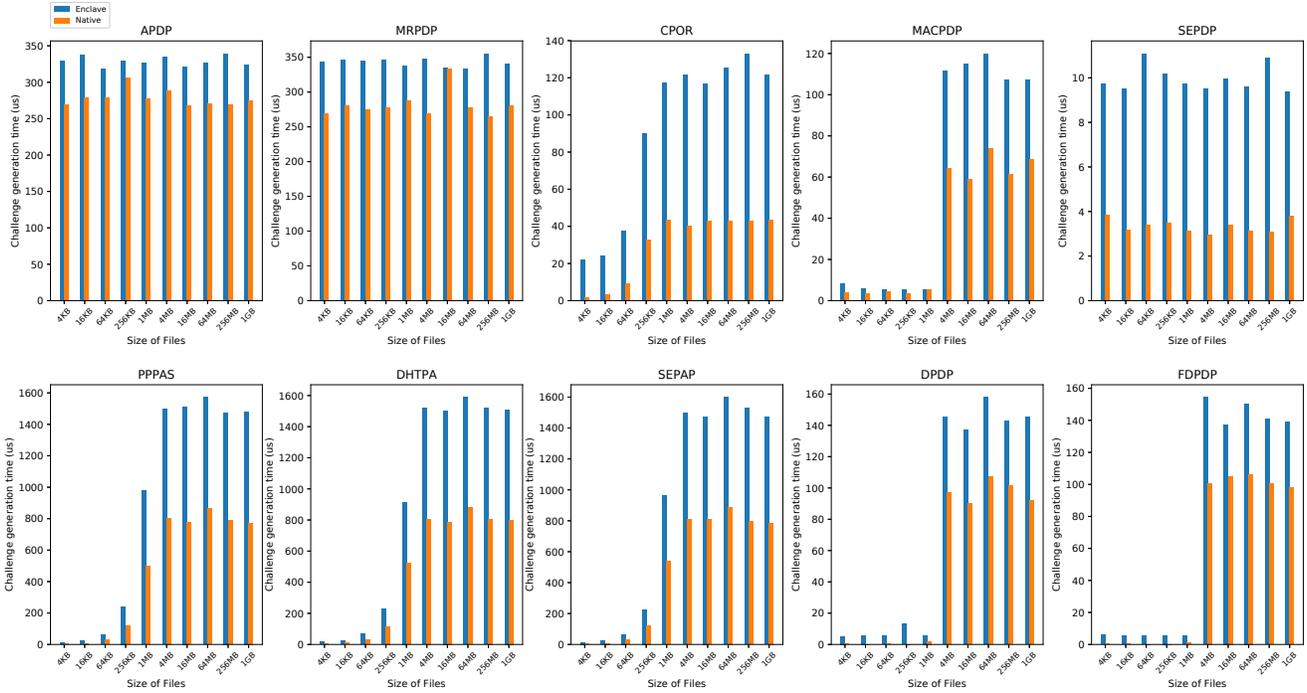


Figure 3: Overhead of Challenge Operations

and MAC-PDP, respectively. The three BLS-based schemes (PPPAS, DHT-PA and SEPAP) saw similar performance degradation, i.e., 34.9% for PPPAS, 36.5% for DHT-PA, and 35.2% for SEPAP, respectively. DPDP and FlexDPDP introduced 47% and 37% overhead respectively, while SEPDP and CPOR experienced 82.0% and 92.2% increase of the verify operation time respectively. The reason for such overhead is similar to that of the overhead of the challenge time described above, the low-overhead operation (the verify operation) is affected more significantly by the `ecall` execution context switch. Though up to 92% runtime overhead, the following experiments (Section 6.4) will demonstrate that such microsecond-range or millisecond-range overhead makes acceptable, or even negligible impact on the throughput for the practical deployment.

Observation 3. Running the verification operation inside the enclave introduces less overhead (12.70%–92.2%) compared with the challenge operation (18.10%–190%), because the challenge operation is relatively “lightweight” compared with the verification operation in terms of computation.

6.4 Evaluation of PDP Request

We measured the response time and throughput of the 10 native PDP schemes and their EnclavePDP implementation, by verifying the integrity of files with different sizes. The response time includes the time of network communication and all operations (i.e., *Challenge*, *Proof*, *Verify*) in the verification phase. Since we set the maximum number of challenge blocks as 460, we intend to choose 1GB file (larger than 460

blocks) and 16KB file (smaller than 460 blocks) to conduct the following experiments.

The right of Table 3 shows the average response time under the condition of the maximum throughput of both native PDP and EnclavePDP on verifying the integrity of 1GB file. “Thr” indicates the number of concurrent threads (imitating multiple users) used to trigger the maximum throughput. As shown in Table 3, the average response time for most of the PDP schemes (including CPOR, SEPDP, MACPDP, APDP, MRPDP, DPDP and FlexDPDP), when implemented in Intel SGX, is almost negligible, with the overhead from 1.0% to 5.4%. In contrast, the overhead of the three BLS-based schemes, i.e., PPPAS, DHT-PA and SEPAP, is 24.5%, 23.4%, and 10.9% respectively. Recall the verification operation for the BLS-based schemes takes significantly longer than that of other PDP schemes, in the scale of second, but the overhead incurred by EnclavePDP is still reasonable.

We conducted an experiment to measure the proportion of challenge/verify time to the total response time, when launching only one thread to issue one PDP request each time. As shown in Table 4, the verification time of the BLS-based schemes accounts for much more proportion than that of other schemes, which well explains why running BLS-based schemes in enclave introduces more overhead compared with other PDP schemes. However, the response time also includes the network communication latency and the time of the proof operation, thus the overhead per PDP request for these PDP schemes is diluted. For most of these 10 PDP schemes, the runtime of the challenge operation and the verify operation accounts for a quite small proportion of the total response time, which is in line with the fact that although per challenge

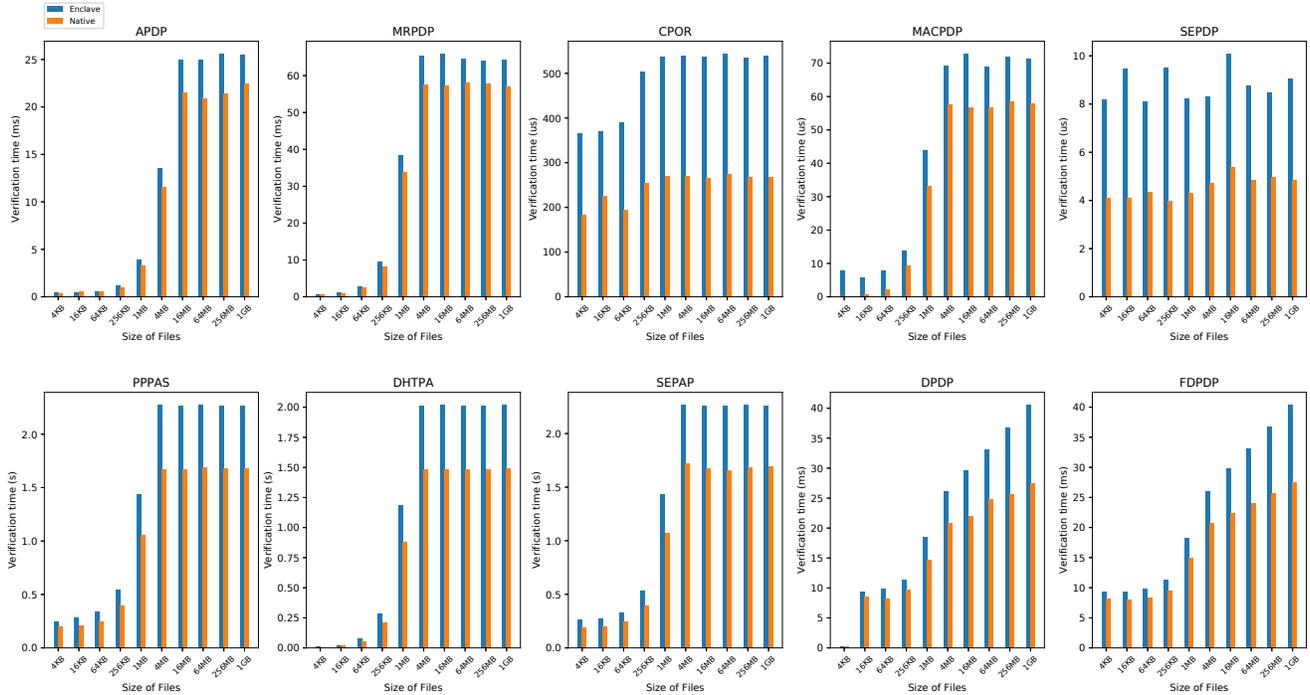


Figure 4: Overhead of Verify Operations

Table 3: Evaluation of PDP Request on 16KB and 1GB File.

Schemes	16KB file				1GB file			
	EnclavePDP	Native PDP	Overhead	Thr.	EnclavePDP	Native PDP	Overhead	Thr.
MACPDP	515 ms (403.2 req/s)	500 ms (418.1 req/s)	3.0% (3.6%)	220	987 ms (239.9 req/s)	971 ms (244.7 req/s)	1.6% (2.0%)	200
APDP [9]	1154 ms (33.8 req/s)	1110 ms (35.1 req/s)	3.9% (3.7%)	40	2164 ms (8.9 req/s)	2079 ms (9.3 req/s)	4.1% (4.5%)	40
MRPDP [11]	957 ms (89.8 req/s)	936 ms (91.1 req/s)	2.2% (1.4%)	100	3115 ms (6.2 req/s)	2976 ms (6.5 req/s)	4.7% (4.6%)	40
SEPDP [10]	642 ms (410.2 req/s)	601 ms (436.7 req/s)	6.8% (6.0%)	275	725 ms (327.6 req/s)	718 ms (334.0 req/s)	1.0% (1.9%)	250
CPOR [12]	539 ms (389.7 req/s)	520 ms (414.2 req/s)	3.6% (6.0%)	250	1140 ms (84.8 req/s)	1131 ms (85.7 req/s)	1.0% (1.1%)	100
DPDP [7]	1095 ms (90.4 req/s)	939 ms (103.4 req/s)	16.6% (12.6%)	120	24655 ms (0.0405 req/s)	23814 ms (0.0418 req/s)	3.4% (3.5%)	5
FlexDPDP [8]	1075 ms (90.7 req/s)	934 ms (104.7 req/s)	15.1% (13.3%)	120	50698 ms (0.052 req/s)	48100 ms (0.0552 req/s)	5.4% (5.5%)	5
PPPAS [19]	8391 ms (3.3 req/s)	6318 ms (4.5 req/s)	32.8% (24.4%)	30	46034 ms (0.363 req/s)	36886 ms (0.465 req/s)	24.5% (21.9%)	20
SEPAP [17]	5552 ms (3.5 req/s)	4162 ms (4.6 req/s)	33.3% (24.0%)	30	41700 ms (0.365 req/s)	37591 ms (0.458 req/s)	10.9% (20.4%)	20
DHT-PA [18]	1311 ms (22.3 req/s)	1110 ms (26.3 req/s)	18.1% (15.2%)	30	34207 ms (0.487 req/s)	27709 ms (0.64 req/s)	23.4% (24.0%)	30

* Note: the value in the "()" is the maximum throughput (req/s) associated with corresponding response time.

* Thr. : Threads indicating concurrent users.

or verify operation introduces relatively high overhead, the impact to per PDP request is almost negligible.

In addition, we find that the proportion of challenge/verify time for most enclave-enabled PDP schemes is in the same order of magnitude as that of native PDP schemes, slightly higher than the latter. For DPDP and FlexDPDP schemes, the higher overhead on challenge time (2.3 times and 1.5 times respectively) might be explained by a loop function (an expensive operation) used to generate non-negative random integers in the challenge generation function of the two enclave-enabled schemes. The 1.6 times overhead on challenge time for the enclave-enabled CPOR scheme is probably due to the extra operations (e.g., private keys loading, challenge backup), which has significant impact on the originally small challenge operation time of the CPOR scheme.

Observation 4. The impact incurred by EnclavePDP to the entire response time, a complete challenge-verify procedure, is acceptable for practical deployment.

From the perspective of maximum throughput, SEPDP,

MAC-PDP and CPOR perform much better than the other schemes. In particular, the maximum throughput of SEPDP and MAC-PDP is one order of magnitude higher than CPOR and two orders of magnitude higher than APDP and MRPDP. This can be attributed to the fact that symmetric encryption (SEPDP and MAC-PDP) is of higher efficiency than asymmetric encryption (e.g., APDP). Meanwhile, the maximum throughput of those three BLS-based schemes (i.e., PPPAS, DHT-PA and SEPAP) is one or several orders of magnitude slower than the above five schemes, since they utilize the BLS signatures primitive to support public auditing at the expense of low efficiency inherited from BLS signatures. The maximum throughput of DPDP and FlexDPDP is another one order of magnitude smaller than the three BLS-based PDP schemes, because building the Rank-Based Authenticated Skiplist (RBASL) or FlexList data structures is not efficient and quite memory-consuming.

Figure 4 shows that the verification time of DPDP and FlexDPDP is about one order of magnitude shorter than that

Table 4: Proportion of Challenge and Verify Time in a PDP Request

		MACPDP	APDP [9]	MRPDP [11]	SEPDP [10]	CPOR [12]	DPDP [7]	FlexDPDP [8]	PPPAS [19]	SEPAP [17]	DHT-PA [18]
Challenge	E	0.015%	0.037%	0.030%	0.001%	0.016%	0.001%	0.001%	0.040%	0.040%	0.050%
	N	0.010%	0.032%	0.025%	0.001%	0.006%	0.0003%	0.0004%	0.026%	0.027%	0.034%
Verify	E	0.011%	2.881%	5.541%	0.001%	0.072%	0.154%	0.210%	62.870%	68.083%	71.942%
	N	0.009%	2.574%	5.104%	0.001%	0.037%	0.108%	0.140%	57.180%	58.753%	63.338%

* Note: "E" means "EnclavePDP"; "N" means "Native PDP".

of the three BLS-based schemes, which seems contradictory to fact that the maximum throughput of the former is about one order of magnitude smaller than that of the latter when verifying the integrity of 1GB file as in the right of Table 3. We conducted another experiment to evaluate the proof generation time of those five schemes to generate proofs on 1GB file. We find that the proof generation time of DPDP is 10s, nearly 5 times of those three BLS-based schemes, i.e., 2.5s for PPPAS, 1.8s for DHTPA and 2.3s for SEPAP, respectively. The proof generation time of FlexDPDP is also about 3 times of those BLS-based schemes. In fact, the proof generation of DPDP and FlexDPDP spends a large amount of time to build RBASL and FlexList, and the property information of the blocks to be checked needs to be sent back to the verifier, which introduces much more communication overhead than that of those BLS-based schemes. Moreover, the size of RBASL and FlexList also depends on the size of the file to be verified. The left part of Table 3 shows that DPDP and FlexDPDP perform better than the BLS-based schemes when verifying the integrity of smaller files.

Observation 5. To support dynamic auditing, the performance of PDP schemes like DPDP and FlexDPDP downgrades significantly, due to the expense of building and managing memory-consuming data structures.

Finally, we also conducted an experiment to evaluate the overhead incurred by EnclavePDP when performing integrity verification on a smaller file, i.e., 16KB. As shown in Table 3, when the number of concurrent threads is the same for 16KB file and 1GB file, verifying 16KB file by EnclavePDP introduces less overhead than 1GB file. For example, with the same 40 concurrent threads, enclave-enabled APDP imposed 3.9% overhead on 16KB file and 4.1% overhead on 1GB file. With the same 30 concurrent threads, enclave-enabled DHT-PA imposed 18.1% overhead on 16KB file, and 23.4% overhead on 1GB file. However, for other schemes, we cannot simply compare the overhead on 16KB file and 1GB file directly, because the number of concurrent threads launched to evaluate the maximum throughput can be quite different, e.g., 120 for DPDP and FlexDPDP on 16KB file and 5 on 1GB file. Overall, when verifying 16KB file, the maximum throughput of EnclavePDP is still in the same order of magnitude as the native PDP, which indicates the overhead caused by EnclavePDP is still acceptable for practical deployment.

7 Related Works

Provable Data Possession Schemes. Many data integrity verification schemes [7–16], [30, 45, 46] have been proposed.

Among them, SEPDP [10], DPDP [7], and FlexDPDP [8] provided support to verify dynamic data. Mirror [16], CPOR [12] and Iris [31] extended PDP schemes to provide data integrity verification with data recovery if any data corruption is identified, i.e., proof of retrievability (POR) schemes. [11, 30] designed the integrity check of static data for multiple copies. PPPAS [19], DHTPA [18] and Qruta [14] proposed privacy-preserving auditing schemes using third parties. Many literature surveys (e.g., [25–29]) presented comprehensive summaries and comparison of the existing PDP scheme by defining a taxonomy of existing PDP schemes. However, these surveys primarily focus on a summary of the existing PDP schemes, without any practical implementation or evaluation of them on real-world cloud storage servers.

Securing Cloud Storage Systems. DEPSKY [47] proposed a cloud-of-clouds storage system, storing data on several cloud services to improve the data integrity and retrievability. Depot [48] designed a cloud storage system to guarantee the consistency of operations on data. It also protects the integrity of data by preventing unauthorized nodes from accessing the data objects. DEPSKY [47] still trusts the cloud storage platforms, while Depot [48] mainly focuses on the consistency and availability of the data. CloudProof [49] used cryptographic keys to create access control policies, which allow users to detect violations of integrity and also prove those violations to a third party. CloudProof mainly aims to provide security guarantees for the SLA (Service Level Agreement) to ensure that users will receive a certain compensation in case of cloud misbehavior.

Intel SGX-based Approaches. LibSEAL [37] presented a secure audit library to detect service integrity violations (e.g., committing operations of Git) by creating non-repudiable audit logs protected by Intel SGX. LibSEAL is implemented as a TLS library, which is not applicable to verify data integrity. EnclaveDB [50] is a secure database that guarantees the confidentiality and integrity of data and queries by placing sensitive data (tables, indexes and other metadata) in Intel SGX enclave. Delegation [51] designed a brokered delegation scheme, which utilizes SGX for users to securely delegate their credentials of service providers to others. Ohrimenko et al [52] rely on SGX to perform privacy-preserving machine learning on collaborative data owned by multi-parties.

8 Conclusion

In order to enable users to independently and confidentially verify the integrity of their outsourced data on cloud storage servers, we present EnclavePDP, a general framework

that utilizes Intel SGX to perform data integrity verification. We tailored Intel SGX SSL library and ported PBC libraries into Intel SGX. Then 10 representative PDP schemes are implemented based on EnclavePDP framework. We deployed EnclavePDP on a real-world cloud application (FastDFS) to evaluate its practicality. The experimental results show that EnclavePDP introduced a reasonable runtime overhead for different sizes of files, thus feasible to be deployed with existing cloud storage services via its convenient interfaces.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by Strategic Priority Research Program of Chinese Academy of Sciences (No.XDC02010900), National Key Research and Development Program of China (No.2016QY04W0903), Beijing Municipal Science and Technology Commission (No.Z191100007119010) and National Natural Science Foundation of China (No.61772078). Peng Liu is supported by NSF CNS-1814679.

References

- [1] Dropbox bug wipes some users' files from the cloud. <https://www.engadget.com/2014/10/13/dropbox-selective-sync-bug/>, 2014.
- [2] Amazon's cloud crash disaster permanently destroyed many customers' data. <https://www.businessinsider.com/amazon-lost-data-2011-4>, 2011.
- [3] Tencent cloud says 'improper operations' led to data loss. <https://www.scmp.com/tech/article/2158785/tencent-cloud-says-improper-operations-led-data-loss-client-it-seeks-implement>, 2018.
- [4] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. Cloudsafetynet: Detecting data leakage between cloud tenants. In *Proc. of ACM CCSW*, 2014.
- [5] A look back: U.s. healthcare data breach trends. https://infosec.uthscsa.edu/sites/default/files/HITRUST_Report-US_Healthcare_Data_Breach_Trends.pdf, 2012.
- [6] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, May 2011.
- [7] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proc. of ACM CCS*, 2009.
- [8] Ertem Esiner, Adilet Kachkeev, Samuel Braunfeld, Alptekin Kupcu, and Ozgur Ozkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. *ACM Transactions on Storage*, 12(4):23, 2016.
- [9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.
- [10] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *Proc. of ACM SecureComm*, 2008.
- [11] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Proc. of IEEE ICDCS*, 2008.
- [12] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proc. of ASIACRYPT*, 2008.
- [13] Łukasz Krzywiecki and Mirosław Kutylowski. Proof of possession for cloud storage via lagrangian interpolation techniques. In *Proc. of NSS*, 2012.
- [14] B. Wang, B. Li, and H. Li. Oruta: privacy-preserving public auditing for shared data in the cloud. *IEEE Transactions on Cloud Computing*, 2(1):43–56, Jan 2014.
- [15] Boyang Wang, Baochun Li, and Hui Li. Knox: privacy-preserving auditing for shared data with large groups in the cloud. In *Proc. of ACNS*, 2012.
- [16] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *Proc. of USENIX Security*, 2016.
- [17] J. Shen, J. Shen, X. Chen, X. Huang, and W. Susilo. An efficient public auditing protocol with novel dynamic structure for cloud data. *IEEE Transactions on Information Forensics and Security*, 12(10):2402–2415, Oct 2017.
- [18] H. Tian, Y. Chen, C. Chang, H. Jiang, Y. Huang, Y. Chen, and J. Liu. Dynamic-hash-table based public auditing for secure cloud storage. *IEEE Transactions on Services Computing*, 10(5):701–714, Sep. 2017.
- [19] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, Feb 2013.

- [20] Intel® software guard extensions programming reference. <https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf>, 2014.
- [21] Intel® software guard extensions ssl. <https://github.com/intel/intel-sgx-ssl>, 2019.
- [22] Gnu multiple precision arithmetic trusted library for intel® software guard extensions. <https://github.com/intel/sgx-gmp>, 2019.
- [23] Pbc library. <https://crypto.stanford.edu/xbc/>, 2019.
- [24] Fastdfs. <https://github.com/happyfish100/fastdfs>, 2013.
- [25] Faheem Zafar, Abid Khan, Saif Ur Rehman Malik, Mansoor Ahmed, Adeel Anjum, Majid Iqbal Khan, Nadeem Javed, Masoom Alam, and Fuzel Jamil. A survey of cloud computing data integrity schemes: Design challenges, taxonomy and future trends. *Computers & Security*, 65:29 – 49, 2017.
- [26] S. G. Worku, Z. Ting, and Q. Zhi-Guang. Survey on cloud data integrity proof techniques. In *Proc. of IEEE AsiaJCS*, 2012.
- [27] Nouha Oualha, Jean Leneutre, and Yves Roudier. Verifying remote data integrity in peer-to-peer data storage: A comprehensive survey of protocols. *Peer-to-Peer Networking and Applications*, 5(3):231–243, Sep 2012.
- [28] Mehdi Sookhak, Hamid Talebian, Ejaz Ahmed, Abdullah Gani, and Muhammad Khurram Khan. A review on remote data auditing in single cloud server: Taxonomy and open issues. *Journal of Network and Computer Applications*, 43:121 – 141, 2014.
- [29] Lei Zhou, Anmin Fu, Shui Yu, Mang Su, and Boyu Kuang. Data integrity verification of the outsourced big data in the cloud environment: A survey. *Journal of Network and Computer Applications*, 122:1 – 15, 2018.
- [30] Ayad F Barsoum and M Anwar Hasan. Integrity verification of multiple data copies over untrusted cloud servers. In *Proc. of IEEE Computer Society CCGRID*, 2012.
- [31] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proc. of ACM ACSAC*, 2012.
- [32] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *Proc. of ASIACRYPT*, 2009.
- [33] Boneh–lynn–shacham. Accessed on April 10, 2019.
- [34] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proc. of NDSS*, 2017.
- [35] S. Gueron. Memory encryption for general-purpose processors. *IEEE Security Privacy*, 14(6):54–62, Nov 2016.
- [36] Innovative technology for cpu based attestation and sealing. <https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing>, 2013.
- [37] Pierre-Louis Aublin, Florian Kelbert, Dan O’Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Libseal: Revealing service integrity violations using trusted execution. In *Proc. of ACM EuroSys*, 2018.
- [38] Elliptic-curve diffie–hellman. https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman, 2019.
- [39] epoll. <https://en.wikipedia.org/wiki/Epoll>, 2019.
- [40] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 213–222. ACM, 2009.
- [41] Intel® digital random number generator (drng) software implementation guide. <https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide>, 2014.
- [42] Intel(r) software guard extensions (sgx) protected code loader (pcl) for linux* os. <https://github.com/intel/linux-sgx-pcl>, 2017.
- [43] Azure confidential computing. <https://azure.microsoft.com/en-us/solutions/confidential-compute/>, 2019.
- [44] Bls signatures: better than schnorr. <https://medium.com/cryptoadvance/bls-signatures-better-than-schnorr-5a7fe30ea716>, 2018.
- [45] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Proc. of Theory of Cryptography Conference*, 2009.
- [46] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proc. of ACM CCSW*, 2009.

- [47] Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *TOS*, 9(4):12:1–12:33, 2013.
- [48] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proc. of USENIX OSDI*, 2010.
- [49] David Molnar, Jay Lorch, , and Raluca Ada and Popa. Enabling security in cloud storage slas with cloudproof. Technical report, May 2010.
- [50] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *Proc. of IEEE SP*, 2018.
- [51] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *Proc. of USENIX Security*, 2018.
- [52] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. of USENIX Security*, 2016.

Robust P2P Primitives Using SGX Enclaves

Yaoqi Jia¹ Shruti Tople² Tarik Moataz³ Deli Gong¹ Prateek Saxena⁴ Zhenkai Liang⁴

¹ACM Member {jiayaoqijia, gnnnnng}@gmail.com ²Microsoft Research t-shtopl@microsoft.com

³Aroki Systems tarik@aroki.com ⁴National University of Singapore {prateeks, liangzk}@comp.nus.edu.sg

Abstract

Peer-to-peer (P2P) systems such as BitTorrent and Bitcoin are susceptible to serious attacks from byzantine nodes that join as peers. Research has explored many adversarial models with additional assumptions, ranging from mild (such as pre-established PKI) to strong (such as the existence of common random coins). One such widely-studied model is the *general-omission* model, which yields simple protocols with good efficiency, but has been considered impractical or unrealizable since it artificially limits the adversary only to omitting messages.

In this work, we study the setting of a synchronous network wherein peer nodes have CPUs equipped with a recent trusted computing mechanism called Intel SGX. In this model, we observe that the byzantine adversary reduces to the adversary in the general-omission model. As a first result, we show that by leveraging SGX features, we eliminate any source of advantage for a byzantine adversary beyond that gained by omitting messages, making the general-omission model *realizable*. Second, we present new protocols that improve the communication complexity of two fundamental primitives — reliable broadcast and common random coins (or beacons) — in the synchronous setting, by utilizing SGX features. Our evaluation of 1000 nodes running on 40 DeterLab machines confirms theoretical efficiency claim.

1 Introduction

Peer-to-peer systems such as BitTorrent [2], Symform [14], CrashPlan [5], StorJ [13], Tor [15] and Bitcoin [1] are becoming popular among users due to ease of accessibility. In such P2P systems, online users can simply volunteer as peers (nodes) to join the network. However, this exact property allows adversarial or Sybil peers to be a part of the network and exhibit a *byzantine* (malicious) behavior. The presence of byzantine adversaries is a major security concern in P2P systems. For example, recently, researchers have demonstrated that in a popular cryptocurrency — Bitcoin — byzantine nodes can collude to eclipse or partition the honest nodes leading to double-spending and selfish mining attacks [50, 67]. Further, byzantine nodes in anonymous P2P networks can become the entry and exit nodes of an honest node’s commu-

nication circuit, by advertising high-bandwidth connections and high-uptimes falsely [19]. These byzantine entry / exit nodes can selectively deny service or severely weaken the core anonymity properties of such systems as Tor, Cashmere and Hydra-Onions [15, 25]. In addition, byzantine nodes in the network can selectively forge, divert, delay or drop messages to disrupt the protocol execution. Therefore, designing robust P2P protocols continues to be an important research problem due to the attacks possible in a byzantine setting.

Researchers have extensively worked in the byzantine model to design solutions for fundamental P2P problems such as reliable broadcast and agreement among the peers [16, 17, 22, 24, 44, 45, 56, 71]. In a quest for efficient protocols that tolerate a larger fraction of malicious nodes, several failure models have been proposed which limit the capabilities of the byzantine adversaries. For instance, one such model is the *general-omission* model where the byzantine node can only omit messages that are either sent or received by it during the execution of a protocol [69, 72]. In this weaker adversarial model, designs with relatively simple and efficient protocols for reliable broadcast tolerating $\frac{N}{2}$ adversarial nodes are known [35, 49, 69, 72]. However, many of these models make strong assumptions, which are not always realistic and have not had a concrete basis for implementation.

Our approach. To this end, we study the possibility of using recent hardware-root-of-trust mechanisms for making previous adversarial models realizable in practical systems. We observe that emerging hardware, specifically Intel SGX, provides stronger trusted computing capabilities, which allow running hardware-attested user-level enclaves on commodity OSes [7–9, 39]. Enclaves provide hardware-isolated execution environment which guarantees that an application executing in an enclave is tamper-resistant and can be attested remotely. Assuming that SGX-like capabilities become commodity and widescale in end hosts, we ask if it is feasible to build robust P2P protocols. Our main observation is that by leveraging the capabilities of such a trusted hardware, one can restrict the behavior of byzantine adversaries to the *general-omission* model in synchronous networks [35, 49, 69, 72].

Specifically, we use four SGX features, i.e., enclave execution (F1), unbiased randomness (F2), remote attestation

(F3) and trusted elapsed time (F4). Based on these hardware features, we enforce six security properties (P1 - P6). First, we enforce execution integrity (P1), message integrity & authenticity (P2) and blind-box computation (P3) to restrict the attacker to not forge messages or deviate from the execution of the given protocol. Thus, the adversarial node can only delay, replay and omit messages. We further leverage lockstep execution (P5) and message freshness (P6) to reduce the adversarial model to the general-omission model, where byzantine nodes have no additional advantage than omitting to send / receive messages. In such model, P3 disallows the adversary to selectively omit messages based on the content. Lastly, the halt-on-divergence (P4) allows us to detect and eliminate peers that selectively omit messages based on identities of senders / receivers, thus in turn reducing round complexity and “sanitizing” the network. Leveraging these properties we can further achieve improvement for the efficiency of protocols. We present efficient designs for reliably broadcasting messages called *Enclaved Reliable Broadcast* (ERB) protocol and an unbiased common random generator called *Enclaved Random Number Generator* (ERNG) protocol. Both ERB and ERNG primitives can be used as building blocks to solve a wide range of problems in distributed systems, such as random beacons [74], voting schemes [65], random walks [48], shared key generation [46, 47], cryptocurrency protocols [61] and load balancing protocols [40, 75].

Results. Our work targets synchronous network where every machine is running an SGX-enabled CPU. Both of our protocols asymptotically reduce the round and communication complexity as compared to previous works in the byzantine model, and match with (or outperform) the results in general-omission model. For a network of size N , the round and communication complexity for ERB are $\min\{f + 2, t + 2\}$ and $O(N^2)$, where t / f ($f \leq t < \frac{N}{2}$) is the number of byzantine peers / peers actually behaving maliciously for one execution of ERB. The communication complexity of the basic ERNG is $O(N^3)$, and the optimized ERNG further reduces the complexity to $O(N \log N)$. We have implemented our solution and the source code is available online [10]. We evaluate both ERB and ERNG, and our experimental results match our theoretical claims.

Contributions. We summarize the main contributions of this paper as below:

- *Realizable General-Omission Model.* We leverage SGX features to reduce byzantine model to general-omission model, where byzantine nodes have no extra advantage than omitting messages.
- *Better Synchronous P2P Protocols.* By enforcing our properties, we can improve the efficiency of P2P protocols. As the first attempt, we propose efficient protocols for reliable broadcast (ERB) and unbiased random number generation (ERNG).
- *Security Analysis & Evaluation.* We provide security analysis and proof for our protocol constructions. Our experi-

mental evaluation confirms the theoretical expectations of our solutions.

2 Problem

Designing efficient solutions for P2P protocols in the byzantine setting is a widely-recognized problem with limited solutions [16, 17, 22, 44, 45, 71]. Our goal is to shed light on how SGX can aid to improve efficiency of synchronous P2P protocols. In this work, we take two fundamental problems as examples: 1) reliable broadcast and 2) common unbiased random number generator.

2.1 Problem Definition

In light of the previous works, we recall the standard definition of *reliable broadcast* [35, 69] and *common unbiased random number* [18] in the synchronous network:

Definition 2.1. (Reliable Broadcast). *A protocol for reliable broadcast in synchronous settings satisfies the following conditions:*

- *(Validity)* If the sender is honest and broadcasts a message m , then all honest nodes eventually accept m .
- *(Agreement)* If an honest node accepts m , then all honest nodes eventually accept m .
- *(Integrity)* For any message m , every honest node accepts m at most once, if m was previously broadcast by the sender.
- *(Termination)* Every honest node eventually accepts a message (m or \perp).

To define a *common unbiased random number generator*, we define the *bias* of any multi-variate function in a standard way [18].

Definition 2.2. (Unbiasedness). *Let $G : \{0, 1\}^{k \times N} \rightarrow \{0, 1\}^k$ be a deterministic multi-variate function that maps N elements in $\{0, 1\}^k$ to one element in $\{0, 1\}^k$. We define the bias of G , $\beta(G)$, as follows:*

$$\beta(G) = \max_{S \subseteq \{0, 1\}^k} \left(\max \left(\frac{E[S]}{E_G[S]}, \frac{E_G[S]}{E[S]} \right) \right),$$

where $E_G[S]$ is the expected number of values in $G(x_1, \dots, x_N) \in S$, and $E[S] = \frac{|S|}{2^k}$, which is the expected value when the output of G is distributed uniformly at random.

Definition 2.3. (Common Unbiased Random Number). *A protocol G generates a common unbiased random number r among N nodes if it satisfies the following conditions with high probability (w.h.p.):*

- *(Agreement)* At the end of the protocol, all the honest nodes agree on the same value r .
- *(Unbiasedness)* The bias of $\beta(G) = 1$.

For the analysis of protocols, we define the following complexities with respect to a single execution of the protocol.

- The *message / communication complexity* is defined as the total number of messages / bits transferred among all nodes in the worst case.
- The *round complexity* is defined as the number of executed rounds (or steps) in the worst-case.

2.2 Attacker Model

We consider a widely-studied standard synchronous model of P2P systems [16, 17, 22, 44, 45, 71]. In this model, our only new requirement is that every peer in the network uses an SGX-enabled CPU to run the P2P protocols. In a network of N nodes, the number of byzantine nodes t is strictly bounded under a fraction of $\frac{N}{2}$. The number of peers that actually behave maliciously for a particular execution of the protocol is $f(\leq t)$. Thus, a P2P network \mathcal{P} is composed of N peers $\mathcal{P} = \{p_1, \dots, p_N\}$ such that $N = 2t + 1$. Every peer p_i in the P2P overlay has an identifier id_i and can communicate with other peers using their ids. The underlying TCP/IP substrate is assumed to provide reliable message delivery within a known bounded delay say Δ . Moreover, we consider a round-based *synchronous* model where each round is equal to the time an honest node requires to send a message and receive a response. Every peer is directly connected to all other peers in the network and knows the network size N . To summarize, we assume: the network size is N (S1); the protocol starts synchronously (S2); the round time is 2Δ (S3); the number of byzantine nodes is limited upto $\frac{N}{2}$ (S4); the peers are connected to each other (S5). This is a prominently used model in the previous literature of distributed P2P systems [16–18, 48, 69, 72].

Our Model using SGX. In our model, a byzantine peer has a compromised or malware-ridden operating system but executes protocols using SGX enclaves [7, 8, 39]. Enclaves guarantee untampered execution in presence of malicious underlying software or co-processes. The byzantine nodes can take arbitrary software actions as long as it does not violate SGX guarantees.

Scope. Our focus is showing how to leverage SGX features to improve the efficiency of synchronous P2P protocols. Our model does not consider an adversary that can perform hardware attacks and break SGX security guarantees. We do not aim to prevent any information leakage through side-channels such as pagefaults, memory accesses or timing attacks to which SGX-enabled CPUs are known to be susceptible [57, 64, 80]. Indeed these problems are under investigation and recent research shows that defending against them is feasible. Existing solutions against these problems can directly apply to our work [62, 68, 76].

2.3 Strawman Solution & Attacks

Consider a strawman protocol for distributed random number generation using reliable broadcast, where the initiator broadcasts a random number m using an initialization message INIT to all the peers in a synchronous network (shown in Algorithm 1). If m is generated randomly and unbiasedly as well as reaches every honest node without being tampered, then all honest nodes will agree on the common unbiased random number m and the goal of the protocol is achieved. In Algorithm 1, upon receiving the INIT message, each peer

Algorithm 1: Strawman distributed random number generation protocol using reliable broadcast.

```

Input: A P2P network  $\mathcal{P}$  composed of  $N$  nodes, an initiator node  $id_{init}$ 
Output: A message  $\hat{m}$ 
1 Initialization:  $\hat{m} \leftarrow \perp$ ;  $S_m \leftarrow \emptyset$ ;  $rnd \leftarrow 1$ 
2 upon self_id is initiator:
3 get( $m$ ) //  $m$  is a random number
4  $\hat{m} \leftarrow m$ 
5 add self_id to  $S_m$ 
6 multicast INIT( $m$ ) to other peers
7 for  $rnd \leq t + 1$  do
8   upon receiving INIT( $m$ ):
9      $\hat{m} \leftarrow m$ 
10    add self_id and sender_id to  $S_m$ 
11    multicast ECHO( $m$ ) to other peers in round  $rnd + 1$ 
12   upon receiving ECHO( $m$ ):
13     if  $\hat{m} = \perp$  then
14        $\hat{m} \leftarrow m$ 
15       add self_id to  $S_m$ 
16       multicast ECHO( $m$ ) to other peers in round  $rnd + 1$ 
17     end
18     if  $m = \hat{m}$  and sender_id  $\notin S_m$  then
19       add sender_id to  $S_m$ 
20       if  $|S_m| = N - t$  then
21         accept  $\hat{m}$ 
22       end
23     end
24    $rnd \leftarrow rnd + 1$ 
25 end
26 if  $rnd > t + 1$  then
27   accept  $\perp$ 
28 end

```

further multicasts an ECHO message to all other peers. After receiving the ECHO messages from the majority of nodes, each peer accepts m as the final message \hat{m} . Note that if the initiator is honest, all honest nodes receive the message INIT during the first round and multicast ECHO messages at the beginning of the second round. In the second round, every honest node receives at least $N - t$ ECHO messages from $N - t$ honest nodes and maybe some byzantine nodes. Thus, after two rounds, every honest node will output the same value m from the initiator, which satisfies all the conditions of reliable broadcast in Definition 2.1. However, we show how a byzantine initiator and other byzantine peers can attack this protocol to violate Definitions 2.1 and 2.3.

Attacks by Byzantine Adversary. Byzantine initiator / peers can tamper with the execution of Algorithm 1 and forge the values of INIT and ECHO messages to perpetrate the following attacks.

A1 (Execution Deviation): For this attack, an adversary deviates from the control flow of the running program for the given protocol. The adversary can disregard essential conditions to jump to the desired instructions and execute them directly. For example, the adversary can skip all the conditions like Line 7 & 13 to directly multicast its ECHO value to parts of honest nodes but not all of them, to introduce equivocation to their final decisions. Moreover, the adversary can also repeat particular instructions to obtain an output she wants. For instance, if m is generated from a random source without being tampered during the execution of the protocol, an unbiased common random number can be agreed among all the peers in the network. A byzantine peer, however, can repeat

the step that generates m (Line 3) from the random source until it returns a favorable random number. Hence, the output is biased as per Definition 2.3.

A2 (Message Forgery): Suppose that the adversary does not deviate from the execution of the given protocol, she can still alter the data flow (including input / output and intermediate states) of the program to forge messages. As per Definition 2.1, a reliable broadcast protocol requires that if one honest node accepts message m then all honest nodes accept m . The adversary can tamper with the INIT and ECHO messages to violate this agreement property of the protocol. A byzantine initiator colluding with other byzantine peers in the network can tamper with Line 6, 11 and 16 in the algorithm such that some honest nodes receive most ECHO messages with m' while others with m . This results in a fraction of honest nodes assigning \hat{m} with m' and accepting m' , while other honest nodes accept m as the final output, thereby causing inconsistency in the network.

A3 (Selective Omission): Assume that the adversary does not deviate from the control flow (i.e., the execution) of the given protocol or tamper with the data flow to forge messages, she can still omit, delay and replay messages in this restricted model. For an omission attack, it has two types: one is based on the content of the transmitted message and the other is dependent on the identity of the sender / receiver. For the first type, the adversary can observe its generated or received random number m and selectively decide to drop or forward it to other nodes based on its value, which introduces a bias in the final output for the honest nodes. For example, if the adversarial peers receive or initiate a message m , which is not the favorable one, they can omit to relay the message to the other nodes, thus all honest nodes may finally agree on \perp instead of m . Further, to violate the agreement condition in Definition 2.1 and 2.3, the adversary can selectively decide to omit the message m depending on whether the destination peer is honest or malicious. It can broadcast m correctly to a few honest nodes and not send the message to the others for the last round. The honest nodes receiving m can multicast m to the others, but the others will not accept it as the execution ends. Thus, the honest nodes that do not receive a message will agree on \perp while others will agree on m .

A4 (Message Delay): Alternatively, to generate an unbiased common random number, every peer can broadcast its random number to all other peers using Algorithm 1. All peers can then XOR the random numbers in the final set to generate the output. To bias this final output, a byzantine peer can intentionally hold its random number until it receives inputs from all other honest peers [18]. In this way, the adversary can “look ahead” in the protocol, calculate the final output and then decide whether to participate in the protocol by sending its random number. If the final random number already favors the adversary then it does not participate in the protocol, otherwise it sends its message to all the peers. Note that, for $t < \frac{N}{2}$, all the byzantine adversaries can collude to introduce

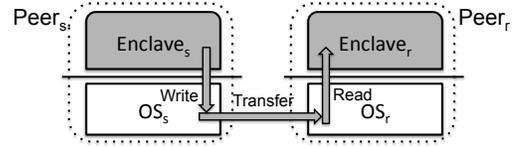


Figure 1: Each peer consists of two entities: an Enclave and an OS. The OS models the operating system and memory. The Enclave models the isolated memory and the secure execution of a program. The sender Enclave_s can send a message via a secure channel to the receiver Enclave_r. The grey areas are secure against malicious OSes of byzantine nodes.

an exponential bias in the final value.

A5 (Message Replay): In the restricted model, the adversarial node can use a message m_{prev} from an instance of the protocol running in parallel, or which was run in the past to one (or more) honest node(s) and forward the correct message m to other honest nodes [59]. This results in an inconsistency where few honest nodes agree on m_{prev} and others agree on m , thereby violating the agreement condition.

3 Solution Overview

In this section, we put forward ideas using SGX features to enforce six security properties to restrict the capabilities (A1 - A5) of a byzantine adversary, as shown in Section 2.

3.1 SGX Features and Security Properties

We first start by recalling Intel SGX features (supported in both simulation and hardware modes in the latest version [7, 9]), which can also be provided by other trusted hardware.

F1: Enclaved Execution - SGX supports hardware-isolated memory region called enclaves such that a compromised underlying OS cannot tamper the execution of the code running inside this enclave.

F2: Unbiased Randomness - SGX provides a function `sgx_read_rand` that executes the `RDRAND` instruction to generate hardware-assisted unbiased random numbers.

F3: Remote Attestation - SGX allows a remote party to verify that an application is running in an enclave on an SGX-enabled CPU.

F4: Trusted Elapsed Time - SGX provides a function `sgx_get_trusted_time` that returns a trusted elapsed time in seconds relative to a reference point.

Abstractly, a peer can be considered as the composition of two entities: an OS and an Enclave as shown in Figure 1. The OS models the untrusted entity including the operating system and memory. It has access to all the system resources such as file system and network. The OS can arbitrarily invoke an enclave program and start its execution. The Enclave models the isolated memory space that loads the program and executes it securely. Thus, Enclave corresponds to the trusted entity of a peer. We illustrate how to enforce P1 - P6 properties using SGX features to thwart A1 - A5 attacks.

P1 (Execution Integrity): With remote attestation (F3), an enclave in one peer can verify the correctness of the running

program for the given protocol on the other nodes and whether it is executing on a valid SGX-enabled CPU or not. Moreover, F1 ensures that the execution in an enclave cannot be tampered with by the OS. F1 and F3 together enforce the execution integrity against A1. Hence, an adversary cannot deviate from the execution of the protocol in an enclave arbitrarily by skipping / repeating instructions to violate the control flow of the running program.

P2 (Message Integrity & Authenticity): In designing our protocols, we first perform a setup phase where each peer connects to every other in the network and then performs a series of steps. Analogous to P1, every enclave first uses F3 to verify the correctness of the protocol executing on other peers. Next, they generate public / private key pairs inside the enclaves and exchange the public keys with each other. Then all the messages transmitted between any two enclaves can be signed to ensure the integrity and authenticity against A2. Moreover, the internal states of the program are also protected using F1. Therefore, the integrity of all messages including input / output / intermediate states is guaranteed. In this case, it is clear that an adversary cannot forge valid messages to bias the honest nodes to make inconsistent decisions.

P3 (Blind-box Computation): F1 ensures that all intermediate states of the protocol's computation are hidden from the OS. Leveraging F2, the provided randomness is also hidden from the OS. This guarantees that the input state is hidden along with the intermediate states of the protocol's execution. We say in this case that the computation is a *blind-box computation*. As the adversarial node does not know the random number and given that the output of the computation is encrypted between the Enclave and the OS, she cannot selectively omit or drop messages based on their contents. Note that an important part of instantiating such a blind-box computation is the ability to instantiate a secure channel between two or more enclaves. In fact, enclaves can agree on a shared key to establish a secure channel using Diffie-Hellman key exchange. Nodes can then encrypt all the messages (including program's intermediate input / output) transmitted between each other to provide confidentiality against malicious OSes. Note that, establishing such a shared key in the enclaved setting is slightly weaker than the standard byzantine model, as the malicious operating system cannot access the shared secret keys and decrypt the exchanged messages due to F1. With P1 - P3, we can reduce the byzantine model to a restricted model, where an adversarial node can only replay, omit and delay messages.

P4 (Halt-on-Divergence): To mitigate selective omission based on nodes' identities (A3), we enforce a security mechanism called halt-on-divergence. This property halts any malicious node deviating from the protocol under some given condition. As an instance, if an adversarial node sends a message, but does not receive adequate responses, it will be forced to leave the current protocol execution. Halt-on-divergence mechanism should be incorporated through a specific ac-

knowledgment protocol instantiation in such a way that every malicious node will be forced to leave if the acknowledgment is not verified. In particular, we introduce an acknowledgment scheme where every receiver acknowledges the sender on receiving every valid message. A message sent over a secure channel is considered valid only if it contains the expected sequence and round number. Naturally, an acknowledgment is not sent for a replayed, omitted or delayed message. Since all honest receivers will reply with acknowledgment (ACK) messages on receiving valid messages, an honest sender should at least receive $t + 1$ ACK messages. Any node receiving less than $t + 1$ ACK messages will halt and leave the network.

The key idea here is to penalize any deviating adversary by churning the node out of the network. This effectively "sanitizes" the network. Thus, to remain a part of the network, every peer should send valid messages to the majority of the network. This property also aids honest nodes in the protocol to decide the final output early and finish the execution immediately.

P5 (Lockstep Execution): F4 allows us to realize a synchronized network across all rounds of a protocol. Each peer uses F4 to decide the correct value of the ongoing round and inserts this round number in all the sent messages. To detect *delay* attacks (A4), a peer simply matches the round number present in an incoming message with the current round number. This defense is hard in the byzantine model with public-key infrastructure even if it supports F1, since the OS can tamper with the relative time to either increase or decrease the rounds of a node. Therefore, having access to a trusted elapsed time functionality allows to perform lockstep execution and detect delay attacks in the restricted model.

P6 (Message Freshness): Similar to [59], we use sequence numbers to ensure message freshness and therefore defend against replay attacks (A5). The main challenge lies in ensuring secure exchange of the initial sequence numbers for each peer and ensuring that the sequence number remains untampered with during the entire intermediate states of the protocol execution. Using the secure channel, the peers securely exchange a nonce or a *sequence number*, which is incremented sequentially by the peer. The nonce is generated using F2 supported by SGX. This prevents the malicious adversary from tampering the initial nonce value to its own advantage. Note that the keys and initial sequence numbers exchange occur only once during the setup phase. If an adversarial node restarts or relaunches its enclave, all the data in the enclave will be removed. Since the enclave does not have the valid sequence number and round number, it cannot re-join the same or any on-going execution, which is equivalent to be considered as a new node for the protocol.

3.2 Overview of Our Results

In this work, we achieve the following results.

R1: *By enforcing (P1 - P6), we reduce the byzantine model to the general-omission model.*

Protocol	Attacker Model	Network Size	Round Complexity	Comm. Complexity
PT [72]	Omission	$t + 1$	$\min\{f + 2, t + 1\}$	$O(N^3)$
PR [69]		$2t + 1$	$\min\{f + 2, t + 1\}$	$O(N^2)$
CT [35]			$2t + 1$	$O(N^2)$
PSL [71]	Byzantine	$3t + 1$	$t + 1$	$O(\exp(N))$
BGP [24]			$\min\{f + 2, t + 1\}$	
BG [22]		$4t + 1$	$t + 1$	
GM [44, 45]		$3t + 1$	$\min\{f + 5, t + 1\}$	$O(\text{poly}(N))$
AD15 [16]			$\min\{f + 2, t + 1\}$	
AD14 [17]	Byzantine	$2t + 1$	$3t + 4$	$O(N^4)$
ERB	Byz. + SGX	$2t + 1$	$\min\{f + 2, t + 2\}$	$O(N^2)$

Table 1: Round complexity and communication complexity for reliable broadcast in synchronous network.

Protocol	Network Size	Round Complexity	Comm. Complexity
AS [18]	$6t + 1$	$O(N)$	$O(N^3)$
AD14 [17]	$2t + 1$	$O(N)$	$O(N^4)$
Basic ERNG	$2t + 1$	$O(N)$	$O(N^3)$
Optimized ERNG	$3t + 1$	$O(\log N)$	$O(N \log N)$

Table 2: Round / communication complexity for random number generation protocols in synchronous distributed systems.

By enforcing P1 - P3, we first reduce byzantine model to a restricted model, in which byzantine nodes can only delay / omit / replay messages. We defer the formalization and proof to Appendix A. We believe that the formalization, while based on traditional cryptographic primitives, provides a new conceptual framing of SGX-enabled CPUs security features, and may be of independent interest. By applying P5 and P6, we further confine the adversarial nodes into the general-omission model.

R2: *We propose an efficient reliable broadcast protocol (ERB) with early stopping, which improves communication complexity from $O(N^3)$ to $O(N^2)$ (refer to Section 4).*

For this result, we leverage four properties. First, P1 - P3 ensure that the adversarial nodes cannot forge messages and deviate from the execution of the protocol. Second, we leverage P4 to show that ERB can broadcast a message to the entire network in $\min\{f + 2, t + 2\}$ rounds with better performance as shown in Table 1.

R3: *We propose a new unbiased random number generation protocol (ERNG) with communication complexity $O(N^3)$ for the basic version, or $O(N \log N)$ for the optimized one, as shown in Table 2 (refer to Section 5).*

With P3 and P5, our unoptimized ERNG solution directly runs our ERB protocol as a sub-routine on the entire network to agree on a random number generated using F2. It has round and communication complexity of $O(N)$ and $O(N^3)$, respectively. We present an optimized version of ERNG by reducing the byzantine fraction from $\frac{N}{2}$ to $\frac{N}{3}$, and forming a cluster of peers within the network. Leveraging the trusted randomness F2 and P3, we can sample a small set of nodes forming a representative cluster. The ERB protocol is executed within this small cluster to generate the final unbiased random number. The round and communication complexity of this optimized ERNG is further reduced to $O(\log N)$ and $O(N \log N)$. Note that the optimized version of ERNG only applies when the size of the network is large enough.

4 Enclaved Reliable Broadcast Protocol

We propose an *enclaved reliable broadcast* (ERB) in the synchronous model using SGX features. The transmitted message, val , between any two peers has the format: $\text{val} := \langle \text{type}, \text{id}, \text{seq}, m, \text{rnd} \rangle$, where $\text{type} \in \{\text{INIT}, \text{ECHO}, \text{ACK}\}$ and rnd represents the current round of the ERB protocol. If $\text{type} = \text{INIT}$, then the initiator peer id_{init} is initiating the broadcast by sending the message m with sequence number seq_{init} at round rnd . If $\text{type} = \text{ECHO}$, it means that its sender knows that id_{init} has sent m , as it has already received either a value with INIT or ECHO for the first time. Finally, if $\text{type} = \text{ACK}$, it means that the peer acknowledges that it has already received either INIT or ECHO values from the sender. We introduce three functions Halt, Multicast and Wait:

- $\text{Halt}(\text{st})$: is a function that sets the state st to \perp .
- $\text{Multicast}(\text{id}_i, \text{val})$: is a functionality that multicasts the value val from the sender p_i to the receiver p_j , for all $j \in [N] \setminus \{i\}$.
- $\text{Wait}(\tau)$: is a function that has as an input the current elapsed time τ in the ongoing round, and suspends the protocol for $(2\Delta - \tau)$ seconds.

Note that Halt function enforces the *halt-on-divergence* property (P4) that we have introduced in Section 3. When the state of the node is set to \perp the node halts on-divergence and is ejected from the P2P network \mathcal{P} . For the sake of exposition, we write $\text{Wait}(\text{rnd})$ in the code description, we say in this case that the protocol waits until the end of the round rnd .

4.1 ERB details

Prior to running the very first instance of the ERB protocol, there is a setup phase. The setup is performed whenever the program (ERB) needs to be updated or changed. We detail the setup phase followed by the explanation of our algorithm.

Setup Phase: Every pair of sender and receiver peer use remote attestation (F3) along with enclaved execution (F1) to verify the correctness of the execution, and therefore enforcing P1 - P3. Then they establish a secure channel using Diffie-Hellman key exchange. This setup enforces P1 - P3, which restricts the byzantine nodes to only omit, replay and delay messages. Next, each peer picks at random a sequence number such that $\text{seq}_s, \text{seq}_r \stackrel{\$}{\leftarrow} \{0, 1\}^k$ and send it to each other. That is, every node has to store the sequence numbers of all other nodes in \mathcal{P} . Finally, every node sets the variable rnd to the value 1. The overhead of the setup is in $O(N^2)$ while the storage overhead per node is in $O(N)$.

Initialization Phase: An initiator node first multicasts the value $\text{val} = \langle \text{INIT}, \text{id}_{\text{init}}, \text{seq}_{\text{init}}, m, \text{rnd} \rangle$, where seq_{init} is the sequence number of the initiator node, and rnd is the round number. The round rnd is first initialized to 1, the enclave will now increment the rnd after every 2Δ seconds—we take advantage of the elapsed time feature of SGX to tie a round to an interval of 2Δ seconds.

Echo Phase: Until round $t + 2$, if a node receives an INIT or ECHO message for the first time, it performs the following actions: (1) start the local clock and initialize the round rnd to 1, the round will increment every 2Δ seconds, (2) if both rnd and seq are consistent with the expected values, it will store the message m , else it just ignores it and treats it as an omitted message. If there is no delay or replay detected, then it multicasts an ECHO message to all nodes at the end of the current round. If the node has already received a valid ECHO message from a distinct node, it will only add the sender's identifier into the set S_{echo} . Recall that at the end of the setup phase, all honest nodes have the same copy of the sequence number of all honest nodes. After every valid instance of the protocol, nodes will increase all sequence numbers by 1.

Decision Phase: If the node has received at least $t + 1$ correct ECHO messages from distinct nodes, i.e., $|S_{\text{echo}}| = t + 1$, then the node accepts \hat{m} . After $t + 2$ rounds, if the node has not received adequate distinct ECHO messages, it accepts $\hat{m} := \perp$. Every multicast requires the node to receive at least $t + 1$ ACK messages, else the node churns out itself using Halt.

4.2 Analysis

In Algorithm 2, if a byzantine sender decides to omit a message, it will not receive a corresponding ACK message as the sent messages never reach the receiver peer. The sender Enclave_s detects that the underlying OS_s is byzantine if it does not receive at least $t + 1$ ACK messages. On failing to receive majority ACK messages, Enclave_s executes the Halt function as per our algorithm and churns itself out of the network based on our halt-on-divergence property (P4). By leveraging the P4, any node can actively detect its own anomalous behavior instead of relying on other nodes to send messages every round to passively identify the anomaly. This results in communication complexity for anomaly detection decreased from $O(N^2)$ to $O(N)$ and the overall complexity is reduced to $O(N^2)$, compared to previous passive-detection approaches, e.g., Perry *et al.*'s work [72]. Here we state our main theorem below and defer the detailed proof to Appendix B.

Theorem 4.1. *If $N \geq 2t + 1$, ERB is a reliable broadcast protocol as defined in Definition 2.1.*

ERB Performance Analysis. Algorithm 2 has a worst-case round complexity equal to $t + 2$ with communication complexity in $O(N^2)$ and $t < \frac{N}{2}$ byzantine nodes. This only occurs if the byzantine peers delay the instance for t rounds before sending the message to at least one honest node. However, in this case, the round complexity is equal to $f + 2$ rather than $t + 2$ as the delay is only in function of the number of byzantine nodes f . On the other hand, byzantine nodes can also decide to not send the message to any honest node, and then the round complexity is $t + 2$ with $O(t)$ communication complexity.

Algorithm 2: ERB: Enclaved reliable broadcast protocol (for a node id_i with the initiator id_{init} sending a message m and a sequence number seq_{init}).

Input: A P2P network \mathcal{P} composed N nodes, a message m and a sequence number seq_{init} for the initiator id_{init}

Output: A message \hat{m}

```

• initialization:  $\hat{m} \leftarrow \perp$ ;  $S_{\text{echo}} \leftarrow \emptyset$ ;  $\text{rnd} \leftarrow 1$ 
• upon  $\text{id}_i = \text{id}_{\text{init}}$  and  $\text{st}_i \neq \perp$ :
   $\hat{m} \leftarrow m$ ;
   $S_{\text{echo}} \leftarrow S_{\text{echo}} \cup \{\text{id}_{\text{init}}\}$ ;
  Multicast( $\text{id}_{\text{init}}, (\text{INIT}, \text{id}_{\text{init}}, \text{seq}_{\text{init}}, m, \text{rnd})$ );
• for  $\text{rnd} \leq t + 2$  do
  • upon receiving  $\langle \text{INIT}, \text{id}_{\text{init}}, \text{seq}, m, \text{rnd}' \rangle$  from  $\text{id}_{\text{init}}$ :
    if  $\text{rnd}' = \text{rnd}$  and  $\text{seq} = \text{seq}_{\text{init}}$  then
      send  $\langle \text{ACK}, \text{id}_{\text{init}}, \text{seq}, H(m), \text{rnd} \rangle$  to  $\text{id}_{\text{init}}$ ;
       $\hat{m} \leftarrow m$ ;
       $S_{\text{echo}} \leftarrow S_{\text{echo}} \cup \{\text{id}_{\text{init}}\} \cup \{\text{id}_i\}$ ;
      Wait( $\text{rnd}$ ) then Multicast( $\text{id}_i, \langle \text{ECHO}, \text{id}_{\text{init}}, \text{seq}, m, \text{rnd} + 1 \rangle$ );
    end
  • upon receiving  $\langle \text{ECHO}, \text{id}_{\text{init}}, \text{seq}, m, \text{rnd}' \rangle$  from peer  $\text{id}_j$ :
    if  $\text{rnd}' = \text{rnd}$  and  $\text{seq} = \text{seq}_{\text{init}}$  then
      send  $\langle \text{ACK}, \text{id}_{\text{init}}, \text{seq}, H(\text{val}), \text{rnd} \rangle$ , where
       $\text{val} = \langle \text{ECHO}, \text{id}_{\text{init}}, \text{seq}, m, \text{rnd} \rangle$  to peer  $\text{id}_j$ ;
      if  $\hat{m} = \perp$  then
         $\hat{m} \leftarrow m$ ;
         $S_{\text{echo}} \leftarrow S_{\text{echo}} \cup \{\text{id}_j\}$ ;
        Wait( $\text{rnd}$ ) then
          Multicast( $\text{id}_i, \langle \text{ECHO}, \text{id}_{\text{init}}, \text{seq}, m, \text{rnd} + 1 \rangle$ );
        end
      if  $\text{id}_j \notin S_{\text{echo}}$  then
         $S_{\text{echo}} \leftarrow S_{\text{echo}} \cup \{\text{id}_j\}$ 
        if  $|S_{\text{echo}}| = N - t$  then
          accept  $\hat{m}$ ;
        end
      end
    end
  • upon Multicast( $\text{id}_i, \text{val}$ ):
    send  $\text{val}$  to  $\text{id}_k$ , for all  $k \in [N] \setminus \{i\}$ ;
    receive  $N_{\text{ack}}$  acknowledgements  $\langle \text{ACK}, \text{id}_{\text{init}}, \text{seq}, H(\text{val}), \text{rnd}' \rangle$ , where
     $\text{rnd}' = \text{rnd}$  and  $\text{seq} = \text{seq}_{\text{init}}$ ;
    if  $N_{\text{ack}} < t$  then
      Halt( $\text{st}_i$ );
    end
  •  $\text{rnd} \leftarrow \text{rnd} + 1$ ;
end
• if  $|S_{\text{echo}}| < N - t$  then
   $\hat{m} \leftarrow \perp$ ;
  accept  $\hat{m}$ ;
end
•  $\text{seq}_{\text{init}} \leftarrow \text{seq} + 1$ ;

```

5 Enclaved Random Number Generation

We present our algorithm that generates an unbiased common random number called *enclaved random number generation* (ERNG).

5.1 Unoptimized ERNG

We detail our unoptimized ERNG in Algorithm 3. At a higher level, every node generates a random number from the enclave, and then performs ERB protocol to broadcast to every node. According to Theorem B.1, all honest nodes in this case will receive the random numbers from all honest nodes after $t + 2$ rounds, and may eventually receive several random numbers from other byzantine nodes. According to the validity requirement, for each ERB instance, every honest node will accept a random number from its initiator or \perp so that all honest nodes have the same final set S_{final} of random numbers. By performing exclusive disjunction (or XOR) of

Algorithm 3: Unoptimized-ERNG: Unoptimized enclaved unbiased random number generation protocol executed by peer p_i .

Input: A P2P network \mathcal{P} composed of N nodes
Output: A unbiased random number r

- initialization: $S_{\text{final}} \leftarrow \emptyset; \text{rnd} \leftarrow 1$
- for** $\text{rnd} \leq t + 2$ **do**
- **if** $\text{rnd} = 1$ **then**
- initiate ERB with inputs $m_i \xleftarrow{\$} \{0, 1\}^k$ and seq_i ;
- end**
- if** $2 \leq \text{rnd} \leq t + 2$ **then**
- execute ERB instances and wait for the output
 ($M_i = \{m_i, \dots, m_i\}$);
- end**
- $\text{rnd} \leftarrow \text{rnd} + 1$;
- end**
- $S_{\text{final}} \leftarrow M_i$;
- $\text{seq}_j \leftarrow \text{seq}_j + 1$, for all $j \in [N]$
- accept $r = \bigoplus_{v \in S_{\text{final}}} v$.

all received random numbers, every honest node obtains an *unbiased common* random number eventually.

Unbiasedness and Randomness Analysis. We describe the main intuition behind the common unbiasedness and randomness of our ERNG’s output and defer formal details to Appendix C. To bias the random value, the adversary may perform several attacks. It can first try to directly forge the random number, however, this is restricted as per execution integrity (P1) and message integrity (P2) enforced by F1 and F3. An adversary can force the program to generate a local random number of its choice. However, each enclave generates an unbiased random number from SGX-enabled CPU instruction `RDRAND` using F2. It is not possible to bias the source of randomness based on the hardware guarantees. Our blind-box computation (P3) together with the secure channel guarantee that an adversary cannot selectively omit its random number based on its value with the goal to bias the output.

One adversarial strategy is to learn the final output and then decide whether to participate or not in the protocol, as in Attack A4. From Algorithm 3, all honest nodes output the final value after round $t + 2$. In order to bias the final value, the adversary should perform the following steps within round number $t + 2$: (1) learn the XOR of random numbers from honest nodes, (2) decide whether to participate or not based on the final value, (3) and multicast its number to honest nodes. In Algorithm 3, the final XOR operation executes only when $\text{rnd} > t + 2$. The execution integrity (P1) ensures sequential execution of our protocol. This property restricts the adversary from directly jumping to the step that computes the XOR operation and learn the result before other honest nodes generate the final output. Next, the lockstep execution (P5) enforced by the elapsed time feature (F4) allows us to bound the time for each round, even on a byzantine peer. Therefore, the adversary cannot look ahead and compute the final output before the last round. If the adversary decides to delay its own random number based on the computed final value, the adversarial random number will be neglected by all honest peers as it will reach after $t + 2$ round. Combining P1, P5 and P3, it is not possible for the byzantine adversary to achieve

steps (1) and (3) simultaneously.

For clarity and without any loss of generality, we model Algorithm 3 as a multi-variate function $G: \{0, 1\}^{k \times N} \rightarrow \{0, 1\}^k$ that maps N elements in $\{0, 1\}^k$ to one element in $\{0, 1\}^k$ such that $G(x_1, \dots, x_N) = \bigoplus_{i=1}^N x_i$.

Theorem 5.1. *The bias of G $\beta(G) = 1$.*

We defer the proof to Appendix C.

5.2 Optimized ERNG

Next, we illustrate the main steps behind our optimized ERNG and defer the pseudo-code details to Appendix D. In this section, we consider that at most $t \leq \frac{N}{3}$ nodes of the network can be byzantine. In this case, ERNG terminates after $\gamma + 4$ rounds, where γ is a statistical parameter. The intuition behind our optimization can be formulated as follows: we first notice that if we select uniformly at random a subset of nodes from \mathcal{P} , we can still guarantee w.h.p. the existence of an honest majority within this smaller representative cluster. By leveraging F2 to generate a random number and blind-box computation (P3), we can sample a set of peers forming a representative cluster. The main remaining question, therefore, is how large this cluster should be. As a starting point, note that if the cluster size is equal to $\frac{2N}{3}$, the probability of having an honest majority is equal to one. This already suggests that the cluster size can be chosen to be smaller. Conceptually, the protocol can be decomposed into three main steps:

Cluster Selection: The purpose of this step is to construct a representative cluster of the entire P2P network. The cluster will consist of nodes selected uniformly at random from \mathcal{P} . At round 1, every node picks uniformly at random a number from $\{0, \dots, \frac{N}{2\gamma} - 1\}$ using SGX (F2). This operation is protected leveraging property P3 in such a way that the computation is hidden from the OS. If the random number equals 0, then the node is *chosen* to be part of the cluster, and then it multicasts a CHOSEN message to all nodes in \mathcal{P} . Upon receiving the CHOSEN message, every chosen node adds the identifier of the sender to its own set S_{chosen} . The size of the set S_{chosen} represents the size of the cluster.

ERB Instances: We first detail a pseudo-solution and then detail our main construction in Algorithm 4 in Appendix D. In round 2, the nodes constituting the cluster will each generate a random number and broadcast it *only* to the nodes constituting the cluster (i.e., peers’ identifiers in S_{chosen}). That is, every node in the cluster will run an independent ERB instance. The intuition behind these multiple instances is the following: for the broadcast to be effective, at least one broadcast instance has to succeed in that the accepted message is different from \perp . However, the complexity of such solution is cubic in $O(|S_{\text{chosen}}|^3)$ which can be a handicap in term of efficiency. As a solution, we incorporate a two-phases clustering. The idea behind this choice is the following: in order to generate a random number we only require one honest node to output a random number r (otherwise the ERNG protocol may out-

put \perp). We can then proceed to select just a few number of nodes to perform the ERB protocol. As long as at least one of these nodes is honest, the correctness of our ERNG holds. Concretely, to generate the second representative cluster, we perform the following: from nodes in S_{chosen} , we uniformly pick at random a value from $\{0, \dots, \gamma' - 1\}$, where γ' is a parameter in function of γ that verifies $\gamma' \leq \gamma$. The peers that output a random number equal to zero will be the only peers able to initiate the ERB protocol. We will show that this strategy will greatly decrease the communication complexity and defer its analysis to Appendix D. Note that this phase lasts for $\gamma + 2$ rounds when all ERB instances terminate.

Selection Decision: At the end of the broadcast phase, the node of the clusters will have each a set containing eventually several random numbers. Note that, as ERB is a reliable broadcast primitive, we know that all honest peers in the cluster will have the same set of random numbers. Once a node in \mathcal{P} receives at least $\gamma + 1$ sets of random numbers, M_K , originating from the nodes in the cluster, it will output the set M_K as S_{final} . All honest nodes will output the same set under the assumption that there is a majority of honest nodes in the cluster. Finally, the random number equals the XOR value of all random numbers in S_{final} .

5.3 Analysis

We present the proofs for the Lemma and Theorems below in Appendix D.

Lemma 5.1. *If up to $t = \frac{N}{3}$ nodes are byzantine, then with at least $1 - \text{negl}(\gamma)$ probability, the representative cluster has more than γ honest nodes, and less than γ byzantine nodes.*

Theorem 5.2. Agreement: *All honest nodes eventually agree on the same common set S_{final} in ERNG.*

Theorem 5.3. Unbiasedness: *The output of the ERNG protocol is an unbiased random number.*

ERNG Performance Analysis. Note that in ERNG, $O(\gamma)$ nodes will be chosen to form the first representative cluster and therefore run $O(\gamma)$ Multicast functions. The communication complexity of this first step is $O(\gamma^2)$. Then, among this first representative cluster, a second cluster will be composed such that all nodes of this cluster will run each an ERB instance. If the size of the second representative cluster is $O(\sqrt{\gamma})$ (as shown in Corollary D.1 in Appendix D), then the communication complexity of this step is $O(\gamma^2 \cdot \sqrt{\gamma})$. Finally, the member of the first representative cluster will multicast the output of the ERB instances to all peers in \mathcal{P} . The communication complexity of this final step is $O(N \cdot \gamma)$. That is, overall, the communication complexity of ERNG equals $O(N \cdot \gamma + \gamma^{\frac{5}{2}})$. Based on Lemmas D.1 and D.2, if N is large such that it verifies $\gamma \in o(N)$, then we can set $\gamma \in O(\log N)$. In this case, the communication complexity and round complexity of ERNG are equal to $O(N \log N)$ and $O(\log N)$.

6 Evaluation

Implementation. We have implemented a prototype of ERB, unoptimized ERNG and ERNG in C/C++ using Intel SGX's Linux SDK [8]. The implementation contains 4030 lines of code (LOC) measured using CLOC tool [4]. Our prototype implementation is open source and available online [10]. We re-use the ported OpenSSL library including cryptographic utilities (`libcrypto` available with Intel SDK), to perform Diffie-Hellman key exchange and AES encryption/decryption. We use `boost` [3] library to implement the communications between any two nodes and use Google `protobuf` libraries [11] and `rapidjson` [12] to serialize data.

Experimental Setup. We use the DeterLab network testbed for our experiments [6]. It consists of 40 servers running Ubuntu 14.04 with dual Intel(R) Xeon(R) hexacore processors running at 2.2 GHZ with 24 cores and 24 GB of RAM. All machines are connected and share the same link with the bandwidth of 128MBps. Every node in our protocol takes up to 1 - 800 MB memory which limits the maximum number of nodes to 2^{10} in our experiments. Due to the limited number of machines in our testbed, we have to run multiple nodes on each machine, thus we use SGX simulation mode¹ for our program and use a simulated Intel attestation service (IAS).

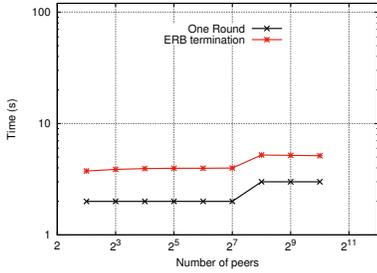
Evaluation Methodology. To evaluate the correctness of our protocols, we measure the round complexity (time to terminate) and communication complexity (network traffic) for ERB, unoptimized-ERNG and ERNG, by varying the number of nodes from 2^2 to 2^{10} . We have highly optimized our system to handle dynamic ports allocations to handle a larger number of nodes within one machine (order of 25 nodes per machine). Part of our results reported in this section are for the *optimistic* case where all nodes behave honestly. We evaluate the round complexity of ERB while varying the number of byzantine nodes in the network up to $\frac{1}{4}$ of the entire network composed of 512 nodes. We also compare our experiment results for the traffic size with theoretical ones to verify if they match our asymptotic analysis.

6.1 ERB Evaluation

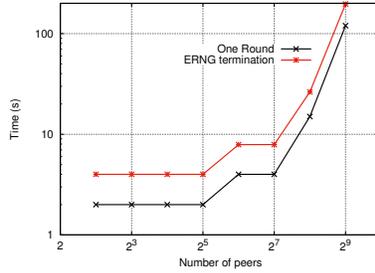
Honest Termination: Constant Scalability. Determining the termination of ERB is essential to validate our reliable broadcast primitive. Fig. (2a) shows that the termination time, in the case of an honest initiator, is nearly equal to twice the value of one round. This validates our theoretical results where we show that ERB finishes in 2 rounds when the initiator is honest. The small increase at 2^8 is purely due to the bandwidth bottleneck of our testbed, as the nodes share the same link.

Traffic Size: Quadratic Scalability. Fig. (3a) demonstrates that the communication complexity quadratically increases in function of the number of peers in \mathcal{P} (note that the x-axis is

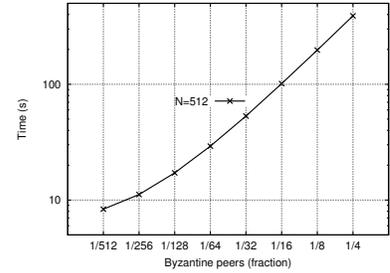
¹ All SGX features we use are supported in the simulation mode and F4 is supported in seconds.



(a) Termination of ERB slightly increase with the number of peers.

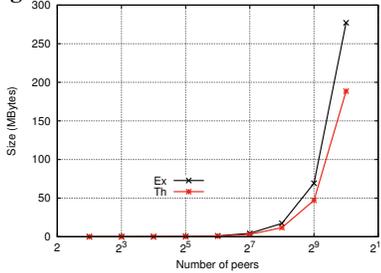


(b) Termination time of ERNG in function of the number of nodes in \mathcal{P} .

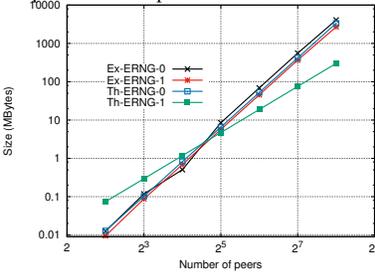


(c) Time termination of ERB linearly increase with the number of byzantine nodes in \mathcal{P} .

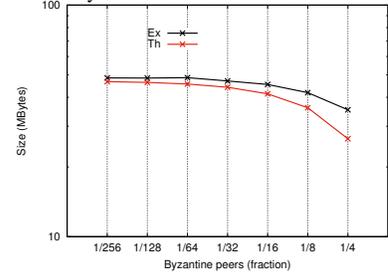
Figure 2: Termination time in seconds for ERB, both unoptimized and optimized versions of ERNG in honest and byzantine network with different fractions.



(a) Communication of ERB in function of the number of nodes in \mathcal{P} .



(b) Communication of ERNG in function of the number of nodes in \mathcal{P} .



(c) Communication of ERB in function of different byzantine peers in \mathcal{P} .

Figure 3: (Th) theoretical and (Ex) experimental comparisons off network overall communication bandwidth in MB for ERB, both unoptimized (ERNG-0) and optimized (ERNG-1) versions of ERNG in honest and byzantine network with different fractions.

logarithmic). The message size of INIT and ACK is around 100 Bytes and 80 Bytes, respectively. For 1024 nodes in \mathcal{P} , the traffic size equals 277 MB. We show that this result matches our theoretical expectation.

6.2 ERNG Evaluation

Honest Termination: Limited Scalability. We show in Fig (2b) that ERNG termination remains slightly constant from 2^2 to 2^7 and then increases afterwards. Unfortunately, this does not reflect our theoretical findings and this is mainly due to the limitation of our testbed, namely, the upper bound on the communication link of 128MBps that all nodes have to share. For small values of peers N , the communication complexity of the unoptimized ERNG is cubic in N , while the optimized version is also (nearly) cubic for smaller values of N . Given a fixed bandwidth, this explains why the termination increases for larger values of N to reach 103 s for one instance.

Traffic Size: Cubic Scalability. Fig. (3b) demonstrates that the communication complexity cubically increases in function of the number of peers in \mathcal{P} for the unoptimized ERNG. Our theoretical results back up our experimental result. For ERNG as the bandwidth links get overflowed much faster, we limited our experiments to 512 nodes. For the optimized ERNG, small values of the number of peers in the network did not allow us to optimally select a cluster size that can guarantee w.h.p. the agreement. In this case, we fix the cluster to be $\frac{2}{3}$ of the network and we show that the traffic size decreases and has a 60% improvement over the unoptimized one. Note that

this result can get much better for a larger number of peers in realistic settings. Here, we draw our theoretical curve for the ideal evaluation which can be guaranteed only for larger N .

6.3 Byzantine case

In Fig (2c), we show that the termination time of ERB linearly increases with the number of byzantine nodes behaving maliciously in the current instance. We gradually increase the fraction of byzantine nodes from $\frac{1}{512}$ to $\frac{1}{4}$. As a strategy of byzantine nodes, we have taken into consideration the worst-case where byzantine nodes create a chain (a byzantine sends its message to only one byzantine node each round and then gets eliminated) in order to delay the termination as much as possible. In the case of $\frac{1}{4}$ byzantine fraction, the ERB termination takes 389 seconds while it only takes 4 seconds in the honest case. For traffic size, if the number of byzantine nodes increases, the communication complexity of ERB decreases as shown in Fig. (3c). This is mainly due to the halt-on-divergence property that will eject the nodes whenever it behaves maliciously. That is when an honest node multicasts a message, the eliminated byzantine node will not acknowledge this message which greatly reduces the communication complexity. For example, for $\frac{1}{4}$ byzantine fraction in a 512-node network, the traffic size equals 35 MB, while in an honest node instance, it is equal to 69 MB, a 50% decrease.

7 Related Work

Reliable broadcast has been extensively investigated in various adversarial models. In our work, we show how Intel SGX

improves the efficiency of existing protocols in these models, renewing interest in studying these protocols with SGX-based implementations.

Reliable Broadcast: Reliable broadcast has been extensively studied since the 1980s, and is closely related to the problem of byzantine agreement (BA). Several excellent surveys on the problem are available [55, 78]. Byzantine agreement can also achieve reliable broadcast [26, 29, 31, 52, 63, 66, 73, 78]. For the asynchronous network, Bracha’s classic reliable broadcast protocol requires $O(N^2)$ communication complexity and tolerates up to $\frac{N}{3}$ byzantine nodes [27, 28]. Cachin and Tessaro [32] leverage erasure codes to improve efficiency and reduce communication complexity. However, as the time is not bounded, messages may incur arbitrary delays, and most protocols do not guarantee terminating runs, except under some special assumptions such as sharing a “common coin” [26, 73].

Lamport *et al.* and Pease *et al.* propose protocols terminating within $t + 1$ rounds and tolerating up to $\frac{N}{3}$ byzantine nodes, but with exponential communication complexity [56, 71]. Berman *et al.* achieve $O(\text{poly}(N))$ communication complexity but only tolerating up to $\frac{N}{4}$ byzantine nodes [22]. Garay *et al.* later present a BA protocol terminating within $\min\{f + 5, t + 1\}$ rounds [44, 45].

To tolerate a larger fraction of byzantine nodes, additional assumptions are often needed. A common assumption is that of having a one-time trusted dealer that pre-deploys PKI in the infrastructure. This assumption, for instance, allows digital signatures to be used for *authentication*, wherein a message claimed to be sent by a node A can be assured to be originating from A [41, 43, 53, 56]. This weakens the capabilities of the byzantine adversary, which cannot forge messages on behalf of honest nodes. Researchers have proposed protocols to use digital signatures to boost the resilience from $\frac{N}{3}$ to $N - 1$, but the communication complexity is still large, i.e., $O(\exp(N))$ and $O(N^3)$ [41, 56]. Katz *et al.* extend the work of Feldman and Micali [42] to employ authenticated channels, and present protocols tolerating $\frac{N}{2}$ byzantine nodes with $O(\text{poly}(N))$ complexity [53]. Fitzi *et al.* also give an authenticated BA protocol that beats this bound ($\frac{N}{2}$) but under specific number-theoretic assumptions [43]. Abraham *et al.* provide a solution with early stopping ($\min\{f + 2, t + 1\}$) and polynomial complexity [16]. In this work, we use SGX features to reduce the byzantine model to the general omission model, and further propose ERB to achieve $\min\{f + 2, t + 2\}$ round complexity and $O(N^2)$ communication complexity.

Researchers also have proposed byzantine fault-tolerant algorithms using trusted services, such as by using trusted computing primitives, primarily focusing on making PBFT more efficient [20, 34, 36–38, 58, 60, 79]. These works have observed similar relation to crash-fault-tolerant protocols, as we have. For example, Chun *et al.* introduce an attested append-only memory (A2M) to remove the ability of adversarial replicas to equivocate without detection, which helps to increase the resilience from $\frac{N}{3}$ to $\frac{N}{2}$ [36]. However, these works have

concentrated on handling asynchronous protocols with weak time assumptions like PBFT. In this paper, in contrast to previous approaches, we work on the round-based synchronous model. Our work extends these ideas to detecting and remediating failures of synchronous network assumptions (e.g. our lockstep execution and halt-on-divergence). Additionally, we investigate the use of our blind-box execution primitive in our new distributed RNG protocol which is bias-resistant, and more efficient using secure sampling for cluster creation. We leave the extension of applying our properties and primitives to asynchronous protocols for future work.

Distributed RNG: Generating common coins in a distributed manner for randomized BA in asynchronous networks can also be used for generating unbiased random numbers [23, 30, 73]. However, these protocols either require a trusted dealer to set up the initial states of different nodes or pre-distribute data to the nodes in the network. Other works employing asynchronous verifiable secret sharing (AVSS) protocols do not have the trusted dealer, but can probabilistically execute with errors [21, 26, 33, 77]. Most of these works employ some cryptographic primitives that, in most case, can be considered heavy-weight and performance unfriendly. Awerbuch *et al.* propose a solution that tolerates up to $\frac{N}{6}$ byzantine nodes, with $O(N)$ round complexity and $O(N^3)$ communication complexity [18] to generate a random number with a constant bias. Other works, such as Andrychowicz *et al.*’s one, generate a common random number based on proof of work [17] with $O(N^4)$ communication complexity, but the output can eventually be biased. Moreover, the large communication cost for most of these approaches prevents scalability to a large number of nodes. We present more efficient (with $O(N \log N)$ communication complexity) and unbiased RNG generation for the synchronous network case.

8 Conclusion

The recent availability of Intel SGX in commodity laptops and servers provides a promising research direction for advancing the area of P2P systems. Our main observation is that leveraging SGX features can restrict a byzantine model to a general-omission model in synchronous systems. We highlight that using SGX we can improve the efficiency of P2P protocols such as reliable broadcast and unbiased random number generator in synchronous settings.

References

- [1] Bitcoin. <https://bitcoin.org/en/>, Accessed: 2020.
- [2] BitTorrent. <http://www.bittorrent.com/>, Accessed: 2020.
- [3] Boost C++ library. <http://www.boost.org/>, Accessed: 2020.

- [4] CLOC. <http://cloc.sourceforge.net/>, Accessed: 2020.
- [5] CrashPlan. <http://www.code42.com/crashplan/>, Accessed: 2020.
- [6] DeterLab. <https://www.isi.deterlab.net/index.php3>, Accessed: 2020.
- [7] Intel Software Guard Extensions. <https://software.intel.com/en-us/sgx>, Accessed: 2020.
- [8] Intel Software Guard Extensions for Linux OS. <https://01.org/intel-softwareguard-extensions>, Accessed: 2020.
- [9] Intel software guard extensions sdk for linux os. https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf, Accessed: 2020.
- [10] P2P using SGX. <https://bitbucket.org/P2PUsingSGX/p2pusingsgx>, Accessed: 2020.
- [11] Protocol Buffers - Google's data interchange format. <https://github.com/google/protobuf>, Accessed: 2020.
- [12] RapidJSON. <http://rapidjson.org/>, Accessed: 2020.
- [13] Storj.io. <http://storj.io/>, Accessed: 2020.
- [14] Symform. <http://www.symform.com/>, Accessed: 2020.
- [15] Tor. <https://www.torproject.org/>, Accessed: 2020.
- [16] I. Abraham and D. Dolev. Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity. In *STOC*, 2015.
- [17] M. Andrychowicz and S. Dziembowski. Distributed cryptography based on the proofs of work. *IACR*, 2014.
- [18] B. Awerbuch and C. Scheideler. Robust random number generation for peer-to-peer systems. In *PODC*, 2006.
- [19] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-resource routing attacks against tor. In *WPES*, 2007.
- [20] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *EuroSys*, 2017.
- [21] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, 1994.
- [22] P. Berman and J. A. Garay. Cloture votes: $n/4$ -resilient distributed consensus in $t+1$ rounds. *STOC*, 1993.
- [23] P. Berman and J. A. Garay. Randomized distributed agreement revisited. In *FTCS*, 1993.
- [24] P. Berman, J. A. Garay, and K. J. Perry. Optimal early stopping in distributed consensus. In *WDAG*, 1992.
- [25] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz. Denial of service or denial of security? In *CCS*, 2007.
- [26] G. Bracha. An asynchronous $[(n-1)/3]$ -resilient consensus protocol. In *PODC*, 1984.
- [27] G. Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 1987.
- [28] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *JACM*, 1985.
- [29] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, 2001.
- [30] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 2005.
- [31] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *DSN*, 2002.
- [32] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *SRDS*, 2005.
- [33] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, 1993.
- [34] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, 1999.
- [35] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *WDAG*, 1990.
- [36] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *OSR*, 2007.
- [37] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *SRDS*, 2004.
- [38] M. Correia, P. Verissimo, and N. F. Neves. The design of a cots real-time distributed security kernel. In *EDCC*, 2002.
- [39] V. Costan and S. Devadas. Intel SGX explained. *IACR*, 2016.
- [40] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *JPDC*, 1989.

- [41] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SICOMP*, 1983.
- [42] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SICOMP*, 1997.
- [43] M. Fitzi and J. A. Garay. Efficient player-optimal protocols for strong and differential consensus. In *PODC*, 2003.
- [44] J. A. Garay and Y. Moses. Fully polynomial byzantine agreement in $t+1$ rounds. In *STOC*, 1993.
- [45] J. A. Garay and Y. Moses. Fully polynomial byzantine agreement for processors in rounds. *SICOMP*, 1998.
- [46] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT*, 1999.
- [47] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology*, 2007.
- [48] R. Guerraoui, F. Huc, and A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC*, 2013.
- [49] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems (2nd Ed.)*, 1993.
- [50] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin’s peer-to-peer network. In *USENIX Security*, 2015.
- [51] Y. Jia, S. Tople, T. Moataz, D. Gong, P. Saxena, and Z. Liang. Robust p2p primitives using sgx enclaves. *IACR*, 2020.
- [52] B. M. Kapron, D. Kempe, V. King, J. Saia, and V. Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *TALG*, 2010.
- [53] J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. In *CRYPTO*, 2006.
- [54] J. Katz and Y. Lindell. *Introduction to modern cryptography*. 2014.
- [55] V. King and J. Saia. Scalable byzantine computation. *SIGACT News*, 2010.
- [56] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *TOPLAS*, 1982.
- [57] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. *arXiv*, 2016.
- [58] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, 2009.
- [59] Y. Lindell, A. Lysyanskaya, and T. Rabin. On the composition of authenticated byzantine agreement. *JACM*, 2006.
- [60] J. Liu, W. Li, G. O. Karame, and N. Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *arXiv*, 2016.
- [61] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *CCS*, 2016.
- [62] S. Matetic, M. Ahmed, K. Kostianen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. Rote: Rollback protection for trusted execution. *IACR*, 2017.
- [63] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *CCS*, 2016.
- [64] Ming-Wei-Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. 2017.
- [65] T. Moran and M. Naor. Split-ballot voting: everlasting privacy with distributed trust. *TISSEC*, 2010.
- [66] A. Mostefaoui, H. Moumen, and M. Raynal. Signature-free asynchronous Byzantine consensus with $t < \frac{n}{3}$ and $O(n^2)$ messages. In *PODC*, 2014.
- [67] K. Nayak, S. Kumar, A. Miller, and E. Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *EuroS&P*, 2015.
- [68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.
- [69] P. R. Parvédy and M. Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *SPAA*, 2004.
- [70] R. Pass, E. Shi, and F. Tramèr. Formal abstractions for attested execution secure processors. *IACR*, 2016.
- [71] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM*, 1980.
- [72] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *TSE*, 1986.
- [73] M. O. Rabin. Randomized byzantine generals. In *FOCS*, 1983.

- [74] M. O. Rabin. Transaction protection by beacons. *JCSS*, 1983.
- [75] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *IPTPS*, 2003.
- [76] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ASIACCS*, 2016.
- [77] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. *IACR*, 2016.
- [78] V. Vaikuntanathan. *Randomized algorithms for reliable broadcast*. PhD thesis, 2009.
- [79] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *TC*, 2013.
- [80] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy*, 2015.

A Primitives and Formal Definitions

In this section, we first start by formally defining the syntax of the communication protocol between two peers, that we denote by Peer channel. Using this definition, we next define various failure modes and primitives. Using SGX, we assume that *execution integrity (P1)* is enforced. We then show that the following properties: *message integrity & authenticity (P2)*, *blind-box computation property (P3)* can be emulated based on the Blinded channel, executing on a particular program. Then we go ahead and formally define the *halt-on-divergence (P4)* property for any program running between two peers. Finally, we show how to reduce the byzantine model to a model where a peer can only replay, omit and delay, dubbed ROD, given that a Blinded channel exists.

Abstractly, a peer can be considered as the composition of two entities: an Enclave and an OS. The OS models the untrusted entity including the operating system and memory. It has access to all the system resources such as file system, network and others. The OS can arbitrarily invoke an enclave program and start its execution. The Enclave models the isolated memory space that loads the program and executes it securely. Thus, Enclave corresponds to the trusted entity of a peer. A concurrent work provides a formal study to show that SGX enclaves can be considered as a trusted entity [70]. The Enclave of the two Peers can interact with each other via their OSs. We formally define a Peer channel as a protocol, $Peer^{ch}$, between a sender $Peer_s = (Enclave_s, OS_s)$ and a receiver $Peer_r = (Enclave_r, OS_r)$. A Peer channel can be seen as a generalization of the traditional secure communication

channel between two parties. The main difference is that the definition of $Peer^{ch}$ protocol is augmented with the program π running within the trusted Enclave.

We define four progressively stronger failure modes: *honest*, *general omission*, ROD and *byzantine* modes of $Peer^{ch}$. Here we introduce a ROD model as an intermediate model, wherein the adversary can only a) Replay b) Omit c) or Delay messages during a protocol, or follow it as prescribed.

We define two primitives: a) Blinded channels and b) halt-on-divergence. Informally, a Blinded channel guarantees confidentiality and integrity of a message over a Peer channel $Peer^{ch} = (Init, Write, Transfer, Read)$.

We show how we build a $Peer^{ch}$ channel using SGX where $Enclave_s$ and $Enclave_r$ are trusted entities. A $Peer^{ch}_{sgx}$ channel is a Blinded channel, and therefore enforces both (P2) and (P3) properties. In particular, we consider that there is a $KeyEx_{\pi}$ protocol between $Enclave_s$ and $Enclave_r$ that is used to generate a session key for a program π . Whenever there is a new program the key has to be re-generated. The key exchange protocol can be instantiated using Diffie-Hellman key exchange, referring to [54] Chapter 9. We use SGX remote attestation to verify that both parties run their code inside an Enclave. While this step is neither required nor captured in the $Peer^{ch}$ definition, it is mandatory to guarantee our *execution integrity (P1)*. Due to space constraints, we defer the formalization and proof to Appendix A in the full version [51].

B ERB Analysis

In this section, we use the same terminology used in Appendix A, namely, we assume that between any two nodes of the network, an $Peer^{ch}_{sgx}$ instantiation of the Blinded Peer channel is enabled. In particular, it provides us with both *message Integrity & authenticity (P2)* and *blind-box computation (P3)* properties. Throughout this section, we implicitly consider that the program is publicly available, and therefore its *execution integrity (P1)* is enforced.

Theorem B.1. *If $N \geq 2t + 1$ where t is the upper bound on the number of byzantine peers, and $Peer^{ch}_{sgx}$ is a Blinded Peer channel, then ERB is a reliable broadcast protocol as defined in Definition 2.1 with worst-case round complexity equal to $t + 2$ and communication complexity equal to $O(N^2)$.*

Proof. Due to space limitation, we defer the proof of the five requirements of terminating reliable broadcast to Appendix C in the full version [51]. Note that the assumption that the peers communicates using $Peer^{ch}_{sgx}$ implies that a byzantine node can only delay, omit or replay messages. As long as the network is synchronous with a fixed time interval for a round to complete, delaying is then equivalent to omitting a message, as the message will not be considered by honest nodes past the round, enforcing therefore the *lockstep execution (P5)* property. Replaying a message is also ineffective as every peer is identified by a sequence number as well,

that is generated by the trusted enclave in the Peer channel, and therefore enforcing the *message freshness* ($P6$) property. Under the assumption that $\text{Peer}_{\text{sgx}}^{\text{Ch}}$ is a Blinded channel, we can replace all occurrences of Multicast by communication between two trusted parties. To sum up, and throughout the proof, it is valid to consider that if there is a delay, omission or replay, this will be equivalent to considering that the first party does not send any message. \square

C Unoptimized ERNG Analysis

In this section, similar to Appendix A, we denote by the ROD mode, a mode where peers in a network \mathcal{P} can only replay, omit and delay messages.

Theorem C.1. *If \mathcal{P} operates in the ROD mode, then the bias of G $\beta(G) = 1$.*

Proof. Note that while G can be modeled as a multi-variate function, it does not capture the sequencing of inputs. For our proof to go through, we need to first show that the sequencing of ERNG is guaranteed and a node can only participate with its input if it starts synchronously with all nodes. For this, we have the following two cases:

- *early start:* if a byzantine node transmits its INIT at $\text{rnd} = 1$, the node outputs (either m or \perp) will be considered as an input for G ,
- *late start:* if a byzantine holds the INIT message until seeing the output, then its input will not be added to S_{final} as the message will be considered delayed. The output of G in this case equals \perp

Note that for both cases, the nodes have to start the protocol at $\text{rnd} = 1$ if they want to participate with their inputs in the final output. Moreover, based on the Blinded channel, we know that nodes can only obtain the final output of G while not viewing any internal state of G , which enforce the *blind-box computation* ($P3$) property. That is, it is valid to consider G as a multi-variate function that is fed all inputs at once. Let us denote by X the random variable that captures the output of G such that $X = X_1 \oplus \dots \oplus X_N$, where X_i 's are random variables that capture the input provided by every node in \mathcal{P} , for all $i \in [N]$. As \mathcal{P} operates in the ROD mode, all honest nodes receive the same set S_{final} at the end of the protocol. We then can rewrite X such that $X = \bigoplus_{i=1}^{\kappa} X_i \oplus \bigoplus_{i=\kappa+1}^N X_i$, where $\kappa = |S_{\text{final}}|$. In the following, we need to show that $E_G[S] = E[S] = \frac{|S|}{2^k}$, for all $S \subseteq \{0, 1\}^k$. Note that $E_G[S] = \Pr[X \in S]$, and therefore it is sufficient to compute $\Pr[X \in S]$.

The second equality follows from the fact that all events are disjoint. Now for a given $x \in S$, $\Pr[X = x] = \Pr[\bigoplus_{i=1}^{\kappa} X_i \oplus \bigoplus_{i=\kappa+1}^N X_i = x] = \frac{1}{2^{s \cdot k}} |\{x_2, \dots, x_{\kappa} \in \{0, 1\}^k\}| = \frac{1}{2^s}$. Thus, $\Pr[X \in S] = \frac{|S|}{2^s}$. This concludes our proof. \square

Algorithm 4: ERNG: Enclaved unbiased random number generation protocol executed by peer p_i .

Input: A P2P network \mathcal{P} composed of N nodes

Output: A unbiased random number r

```

• initialization:  $S_M \leftarrow \emptyset; S_{\text{final}} \leftarrow \emptyset; S_{\text{chosen}} \leftarrow \emptyset; \text{rnd} \leftarrow 1$ 
for  $\text{rnd} \leq \gamma + 4$  do
  if  $\text{rnd} = 1$  then
    • every peer  $p_i$  compute  $r_i \xleftarrow{\$} \{0, \dots, \frac{N}{2\gamma} - 1\}$ ;
    if  $r_i = 0$  then
      Multicast( $\text{id}_i, \text{val}$ ), where
       $\text{val} = \langle \text{CHOSEN}, \text{id}_i, \text{seq}_j, \perp, 1 \rangle$ ;
       $S_{\text{chosen}} \leftarrow \{\text{id}_i\}$ ;
    end
    • upon receiving  $\text{val} = \langle \text{CHOSEN}, \text{id}_j, \text{seq}_j, m_j, \text{rnd}_j \rangle$ 
      if type = CHOSEN and  $\text{rnd}_j = 1$  and  $\text{seq}_j = \text{seq}_i$  then
         $S_{\text{chosen}} \leftarrow S_{\text{chosen}} \cup \{\text{id}_j\}$ ;
      end
    end
  • if  $r_i = 0$  and  $\text{rnd} = 2$  then
    compute  $r'_i \xleftarrow{\$} \{0, \dots, \gamma - 1\}$ ;
    if  $r'_i = 0$  then
      initiate ERB with inputs  $m_i \xleftarrow{\$} \{0, 1\}^k$ ,  $\text{seq}_i$  and
      peers in  $S_{\text{chosen}}$ ;
    end
     $\text{seq}'_j \leftarrow \text{seq}_j$ , for all  $\text{id}_j \in S_{\text{chosen}}$ ;
    end
  if  $r_i = 0$  and  $3 \leq \text{rnd} \leq \gamma + 2$  then
    execute ERB instances and wait for the output;
  end
  if  $r_i = 0$  and  $\text{rnd} = \gamma + 3$  then
    Wait( $\text{rnd}$ ) then obtain  $M_i = \{\hat{m}_1, \dots, \hat{m}_i\}$ ;
     $\text{seq}'_j \leftarrow \text{seq}'_j$ , for all  $\text{id}_j \in S_{\text{chosen}}$ ;
  end
  if  $\text{rnd} = \gamma + 4$  then
    • if  $r_i = 0$  then
       $S_M \leftarrow S_M \cup \{M_i\}$ ;
      Multicast( $\text{id}_i, \langle \text{FINAL}, \text{id}_i, M_i, \text{seq}_j, \gamma + 4 \rangle$ );
    end
    • upon receiving  $\text{val} = \langle \text{FINAL}, M_j, \text{seq}'_j, \text{rnd}_j \rangle$ 
      if  $\text{rnd}_j = \gamma + 4$  and  $\text{seq}'_j = \text{seq}_j$  then
         $S_M \leftarrow S_M \cup \{M_j\}$ ;
        if # of  $M_{\kappa} \geq \gamma + 1$  where  $M_{\kappa} \in S_M$  then
           $S_{\text{final}} \leftarrow M_{\kappa}$ ;
          accept  $r = \bigoplus_{v \in S_{\text{final}}} v$ .
        end
      end
    end
  end
   $\text{rnd} \leftarrow \text{rnd} + 1$ ;
end
•  $\text{seq}_j \leftarrow \text{seq}_j + 1$ , for all  $j \in [N]$ ;

```

D Optimized ERNG

We present a pseudo-solution of our optimized ERNG in Algorithm 4.

Lemma D.1. *If up to $t = \frac{N}{3}$ nodes are byzantine, then with at least $1 - \text{negl}(\gamma)$ probability, the representative cluster has more than γ honest nodes, and less than γ byzantine nodes.*

Proof. In ERNG at round 1, every node picks uniformly at random a value from $\{0, \dots, \frac{N}{2\gamma} - 1\}$. That is, every node has a probability equal to $q = \frac{2\gamma}{N}$ to be chosen as a representative.

Let H_i and B_i be two random variable that equal 1 if the i^{th} honest and byzantine node is chosen respectively, otherwise they equal zero. Let us denote by $H = \sum_{i=1}^{2t} H_i$ and $B = \sum_{i=1}^t B_i$ the number of selected honest and byzantine nodes in the cluster. Then both H and B are distributed following a binomial distribution with a number of trials equal to $2t$ and t , respectively. We have $E[H] = \sum_{i=1}^{2t} E[H_i] = 2t \cdot \frac{2\gamma}{N} = \frac{4t\gamma}{N}$. Similarly, $E[B] = \frac{2t\gamma}{N}$. Based on two variations of Chernoff bound, considering $t = \frac{N}{3}$, we obtain that

$$\Pr[H > (1 - \delta_1) \frac{4\gamma}{3}] \geq 1 - e^{-\frac{2\delta_1^2\gamma}{3}},$$

similarly, $\Pr[B < (1 + \delta_2) \frac{2\gamma}{3}] \geq 1 - e^{-\frac{2\delta_2^2\gamma}{9}}$, where $\delta_1, \delta_2 < 1$. For a choice of $\delta_1 = \frac{1}{4}$ and $\delta_2 = \frac{1}{3}$, we obtain,

$$\Pr[H > \gamma] \geq 1 - e^{-\frac{\gamma}{24}},$$

and,

$$\Pr[B < \gamma] \geq 1 - e^{-\frac{\gamma}{4t}}.$$

□

Lemma D.2. *If $\gamma' = \sqrt{\gamma}$, then the probability that $\Omega(\sqrt{\gamma})$ honest nodes are selected to be in the second representative cluster is at least $1 - \text{negl}(\gamma)$.*

Proof. Based on Algorithm 4, every node in the cluster has a probability of $\frac{1}{\gamma}$ to be chosen. Let us denote by X_i the random variable equal to one if the node is selected. We then denote by, $H' = \sum_{i=1}^H X_i$, the random variable that counts the number of honest node in the second cluster. Based on Wald's equation, we obtain $E[H'] = \frac{E[H]}{\gamma} = \frac{4\gamma}{3\gamma}$. Then, based on Chernoff bound, we obtain for $\delta < 1$,

$$\Pr[H' > (1 - \delta) \cdot \frac{4\gamma}{3\gamma}] \geq 1 - e^{-\frac{4\delta^2\gamma}{3\gamma}}$$

if we set $\delta = 1 - \frac{1}{\gamma}$ and $\gamma' = \sqrt{\gamma}$, then we obtain

$$\Pr[H' > \frac{4\sqrt{\gamma}}{3}] \geq 1 - e^{-\sqrt{\gamma}}.$$

This ends out proof. □

Note that we can obtain better bounds if we consider computing the pmf of H' as it follows a binomial distribution with a binomial number of trials

Corollary D.1. *If $\gamma' = \sqrt{\gamma}$, then the size of the first and second representative clusters is in $O(\gamma)$ and $O(\sqrt{\gamma})$ w.h.p*

The proof of the corollary directly follows from Lemma D.2.

Theorem D.1. Agreement: *All honest nodes eventually agree on the same common set S_{final} in ERNG.*

Proof. In round 1, $|S_{\text{chosen}}|$ nodes are uniformly at random selected to be part of the representative cluster. Based on Lemma D.1, we have shown that the cluster contains *strictly* more than γ honest nodes, and *strictly* less than γ byzantine nodes w.h.p. when $t < \frac{N}{3}$. That is, we have created a new smaller P2P network S_{chosen} in which the honest nodes represent the majority. In the cluster, all honest nodes know each other, but byzantine nodes may deliberately not contact honest nodes on purpose. In this case, the cluster will be more robust with less byzantine nodes. Thus, all the results introduced for ERB will hold for this cluster of nodes.

From round 2 to round $\gamma + 3$, the second cluster has more than $\sqrt{\gamma}$ honest nodes w.h.p. according to Lemma D.2. For each instance of ERB— whether initiated by an honest or byzantine node, the honest representative nodes will agree on a same message. Since there is at least one honest sender, all honest nodes will accept the honest sender's message for its run of ERB. After around $O(\sqrt{\gamma})$ runs, all honest nodes will agree on the same set of random numbers. Since the number of honest representative nodes is larger than γ and all of them will multicast FINAL messages for the same set of messages in round $\gamma + 4$, then all honest nodes will receive adequate FINAL messages to accept the common set S_{final} . □

Theorem D.2. Unbiasedness: *The output of the ERNG protocol is an unbiased random number.*

sketch. Given Theorem D.1, we know that all honest nodes agree on the same set S_M . On the other hand, leveraging Peer_{SGX}^{Ch} Peer channel, we know that all random numbers in the ERNG protocol are generated within the SGX enclave and never tempered with as the network is in the ROD model. Finally, it is sufficient to show that if all random numbers generated in SGX are random then the output of ERNG is an unbiased random number, which holds given SGX primitive generates unbiased random number against the operating system according to Theorem C.1. □

In ERNG, since the message m_i is a random number generated by the SGX and proposed by the peer p_i , then eventually every honest node accepts the same set S_{final} of random numbers according to Theorem D.1. By performing exclusive disjunction (or XOR) of all the random numbers in S_{final} , every honest node can obtain a common random number r . In the meantime, the random number r is unbiased against byzantine nodes.

aBBRate: Automating BBR Attack Exploration Using a Model-Based Approach

Anthony Peterson
Northeastern University

Samuel Jero*
Purdue University

Endadul Hoque
Syracuse University

David Choffnes
Northeastern University

Cristina Nita-Rotaru
Northeastern University

Abstract

BBR is a new congestion control algorithm proposed by Google that builds a model of the network path consisting of its bottleneck bandwidth and RTT to govern its sending rate rather than packet loss (like CUBIC and many other popular congestion control algorithms). Loss-based congestion control has been shown to be vulnerable to acknowledgment manipulation attacks. However, no prior work has investigated how to design such attacks for BBR, nor how effective they are in practice. In this paper we systematically analyze the vulnerability of BBR to acknowledgment manipulation attacks. We create the first detailed BBR finite state machine and a novel algorithm for inferring its current BBR state at runtime by passively observing network traffic. We then adapt and apply a TCP fuzzer to the Linux TCP BBR v1.0 implementation. Our approach generated 30,297 attack strategies, of which 8,859 misled BBR about actual network conditions. From these, we identify 5 classes of attacks causing BBR to send faster, slower or stall. We also found that BBR is immune to acknowledgment burst, division and duplication attacks that were previously shown to be effective against loss-based congestion control such as TCP New Reno.

1 Introduction

BBR (Bottleneck Bandwidth and Round-trip propagation time) is a new congestion control algorithm for TCP [24] and QUIC [25] proposed by Google in 2016. BBR is motivated by how commonly deployed *loss-based* congestion control algorithms inaccurately rely on packet loss as the primary signal for network congestion, often leaving networks underutilized or highly congested. This inaccuracy occurs because in today's networks, the relationship between packet loss and network congestion has become disjoint due to varying switch buffer sizes. Instead, BBR is *model-based*, as it creates a model of the network by periodically estimating the

available bottleneck bandwidth B_{t1Bw} and round-trip propagation delay RT_{prop} , which are used to govern the rate packets are sent into the network and the maximum amount of data allowed in-transit.

Prior work [29, 30, 32, 36] showed how loss-based congestion control algorithms (*e.g.*, New Reno, CUBIC) designed for TCP are prone to *acknowledgment manipulation attacks*, where an adversary exploits the semantics of acknowledgments to mislead the sender (*i.e.*, the victim) of a flow about network congestion. These attacks are possible because TCP headers are unencrypted and have no authentication mechanism other than a random initial sequence number which may be observed or predicted by *on-path* [29] or *off-path* [7, 23, 35] attackers, respectively. While at first it may appear BBR is less prone to such attacks, as it relies on a different congestion control approach, its estimation of B_{t1Bw} and RT_{prop} is based on received acknowledgments. The impact of such attacks can not be easily assessed from existing attacks against loss-based congestion control, because BBR follows a different algorithm for adjusting its sending rate. Given BBR is implemented for TCP [8], the underlying protocol for much of the Internet traffic, and being deployed on YouTube and Google.com [9], studying BBR security and its vulnerability to acknowledgment manipulation attacks is critical.

In this work, we discover and analyze acknowledgment manipulation attacks targeting the Linux TCP BBR congestion control implementation, a popular implementation of BBR. We use a protocol-fuzzing approach to systematically inject at runtime maliciously modified acknowledgment packets that target the core mechanism of BBR: the estimation of B_{t1Bw} and RT_{prop} . In order to achieve this, we adapt TCPWN¹, a TCP congestion control protocol fuzzer, to automatically find vulnerabilities targeting BBR. TCPWN attack strategies are defined by tuples that dictate which type of acknowledgment manipulation attack to execute when the sender is in a certain congestion control state. TCPWN uses the model of the congestion control algorithm to map all theoretically possible

*Samuel Jero is now at MIT Lincoln Laboratory. This work was done while at Purdue University.

¹<https://github.com/samueljero/TCPwn>

attack paths to actual attack strategies. It then uses a state inference algorithm by observing network traffic to discern when to inject the counterfeit acknowledgments. Since TCPWN supports only loss-based congestion control algorithms, we derive a finite state machine for BBR by consulting documentation [9, 16, 17], presentations [10–15] and source code [8]. We additionally develop a new algorithm for inferring the current BBR state in real-time based on network traffic alone, and integrate it with TCPWN.

Using this approach, we automatically generated and executed 30,297 attack strategies from both *off-path* and *on-path attackers*, of which 8,859 caused BBR to send data at abnormal rates: 14 caused a faster sending rate, 4,025 caused a slower sending rate and 4,820 caused a stalled connection (*i.e.*, the flow did not complete). All of these successful attacks originated from an *on-path attacker* with read/write access to the flow. Attacks causing slower/stalled sending performance could be used by an adversary to throttle other flows—leading to poor performance for victim flows and possibly making more bandwidth available to the attacker’s flows. Those causing faster sending performance could be used by a destructive adversary to increase network congestion, leading to unfairness, poor quality of service and congestion collapse. Attacks causing stall connections are a form of denial of service attacks difficult to detect as the connection is active and data is being sent, but with no progress for the flow itself. We summarize our contributions as follows:

- We derive the first *state machine model* for BBR and use it to demonstrate that BBR is vulnerable to acknowledgement manipulation attacks.
- We derive an algorithm for estimating the current BBR state in real-time by observing network traffic.
- We adapt a TCP congestion control fuzzer, TCPWN, to BBR using our newly derived BBR state machine and inference algorithm to automatically generate and execute 30,297 automatically attack strategies.
- We identify 5 classes of acknowledgement manipulation attacks from *on-path attackers* against BBR that cause faster, slower and stalled sending rates. We did not find effective attacks from *off-path attackers*. To the best of our knowledge, we are the first to discover and evaluate attacks on BBR.
- We analyze how BBR distinctly reacts to these attacks, in comparison with other congestion control algorithms. We also found that BBR is immune to acknowledgment burst, division and duplication attacks that were previously shown to be effective against loss-based congestion control such as TCP New Reno.

2 Vulnerability of BBR to Attacks

We now describe BBR, derive a model for it, and show how an attacker can exploit the model to create attacks.

Algorithm 1 Delivery rate samples [17] are computed to estimate the bottleneck bandwidth. For each new ACK, the average ACK rate is computed between when a data segment is sent to when an acknowledgment is explicitly received for it. Delivery rates are capped by the send rate as data should not arrive at the receiver faster than it is transmitted.

Input: A data segment P and a BBR connection C .

Output: The delivery rate sample

```

1: function COMPUTEDELIVERYRATESAMPLE(P, C)
2:   data_acked = C.delivered - P.delivered
3:   ack_elapsed = C.delivered_time - P.delivered_time
4:   send_elapsed = P.sent_time - P.first_sent_time
5:   ack_rate = data_acked / ack_elapsed
6:   send_rate = data_acked / send_elapsed
7:   delivery_rate = min(ack_rate, send_rate)
8:   return delivery_rate
9: end function

```

2.1 BBR Overview

BBR is motivated by how *loss-based* congestion control algorithms such as CUBIC and New Reno assume packet loss implies network congestion, which is not always the case. As a result, sending behavior is adjusted based on signals possibly unrelated to actual congestion, leading to network underutilization and excessive queue delay (bufferbloat).

Instead of relying solely on packet loss to infer congestion, BBR is *model-based* meaning congestion is inferred primarily by two properties of the network path: its bottleneck bandwidth $B_{t}lBw$ and round-trip propagation delay RT_{prop} . BBR paces its sending rate proportionally to $B_{t}lBw$ and aims for at least one $BDP = B_{t}lBw \times RT_{prop}$ worth of data in-flight for full utilization. At any given time, its sending rate is limited by two factors: the congestion window $cwnd$, or $pacing_rate = pacing_gain \times B_{t}lBw$ that defines inter-packet spacing. Pacing, first proposed by Zhang *et al.* [39], aims to reduce burstiness and in some situations, offers improved fairness and throughput [2]. BBR caps $cwnd$ to $2 \times BDP$ to overcome delays in received acknowledgments, which would otherwise cause BBR to underestimate the bottleneck bandwidth [10]. Obtaining an accurate and up-to-date model of the network path is essential to BBR’s effectiveness, and thus the model is updated on every new acknowledgement.

2.2 Estimating the Network Path Model

Accurate measurements of $B_{t}lBw$ and RT_{prop} are obtained sequentially at different times and network conditions because the network conditions required to obtain accurate measurements of each parameter interfere with each others measurements. At mutually exclusive times, BBR adjusts its sending rate so the network conditions are met for each parameter. For $B_{t}lBw$, the sending rate is increased to discover available bottleneck bandwidth while for RT_{prop} , the congestion window is reduced to 4 packets. Note that increasing sending rate to measure $B_{t}lBw$ may create queues which would create inaccurate RT_{prop} measurements. Decreasing the sending

rate to measure RT_{prop} would not allow available bandwidth to be discovered.

Bottleneck Bandwidth. The bottleneck bandwidth is estimated by employing a max filter that retains the maximum observed delivery rate sample over the past 10 round-trips. A delivery rate sample is computed on each new ACK, which is shown in Algorithm 1. Delivery rate samples represent the average acknowledgment rate between when a data segment is transmitted to when an acknowledgment is received for that segment. Delivery rate samples are only computed for the exact packet it acknowledges. This is because factors such as delayed acknowledgment can cause delivery rates to be overestimated. These samples are used primarily to estimate the rate at which data is arriving at the receiver, which is naturally capped by the bottleneck bandwidth. Delivery rate samples only reflect the actual bottleneck bandwidth when BBR sends at a rate that matches or exceeds capacity, which is accomplished by periodically increasing its sending rate 25% faster than the current B_{tLBw} .

Round-Trip Propagation Delay. BBR estimates RT_{prop} using a min filter that retains the minimum observed round-trip time sample over the past 10 seconds. Round-trip samples are measured by computing the elapsed time between when a flight of data is sent to when it is acknowledged. Accurately measuring the RTT presents a challenge because packets queued in switch buffers cause increased and inaccurate RTT samples. To overcome this, BBR drains switch queues by periodically limiting its sending behavior. After measuring B_{tLBw} , it is entirely possible the bottleneck is already saturated, causing queue build-up. To mitigate this, BBR sends 25% slower than the current estimated B_{tLBw} immediately after measuring the bottleneck link. BBR also reduces $cwnd$ to 4 packets every 10 seconds to update RT_{prop} , which we describe in the following section in greater detail.

Rate Limiting. BBR attempts to detect token-bucket policers (TBPs) as they can cause data to be sent faster than the token drain rate, leading to high packet loss. In networks with such TBPs, it is common to see bursts of throughput before tokens are exhausted, after which packets are dropped. Due to BBR's long-lived B_{tLBw} max filter, the burst rate would cause the estimated bottleneck bandwidth to be greater than the token drain rate, leading to high packet loss for as long as 10 round-trips. BBR detects TBPs when there is significant packet loss and consistent throughput, after which it paces its sending rate to the estimated token drain rate for 48 RTTs.

2.3 A State Machine for BBR

To systematically analyze BBR, we derive a finite-state machine (FSM) for it. To the best of our knowledge no such model has been published, so we empirically developed our own through documentation [9, 16, 17], presentations [10–15] and source code [8]. In Figure 1, we illustrate our BBR FSM and describe its variables and events in Table 1.

BBR employs several similar mechanisms to traditional congestion control algorithms. (Readers can refer to Appendix E to revisit the background on congestion control). When a flow first begins, BBR uses a mechanism to quickly discover the available bandwidth (*i.e.* slow start). Afterwards, BBR paces its sending rate at the estimated bandwidth (*i.e.* congestion avoidance) while simultaneously probing the network for available bandwidth and updating its network path model: B_{tLBw} and RT_{prop} . Even though packet loss is not at BBR's core, BBR includes mechanisms to handle such cases. Finally, BBR includes methods for detecting and accounting for token-bucket policers, as they can allow traffic bursts until tokens run up, making BBR to send too quickly causing packet loss. The states of our BBR FSM are:

Startup. Similar to slow start, Startup is the first state BBR enters and aims to quickly discover the available bottleneck bandwidth by doubling its sending rate on each round-trip. Startup transitions into Drain when either $cwnd$ reaches $ssthresh$ or if three consecutive delivery rate samples show less than a 25% increases over the last, indicating the bottleneck bandwidth has been reached.

Drain. This state aims to drain queues that were likely created during Startup. Those queues are reduced in a single round-trip by sending data at $\ln(2)/2 \approx 0.34$ times the rate before entering this state, after which ProbeBW is entered.

ProbeBW. Similar to congestion avoidance, ProbeBW aims to pace the sending rate at the estimated bottleneck bandwidth, achieve fairness, and probe for additional bandwidth with low queuing delay. These are accomplished using gain cycling where the $pacings_gain$ cycles through a set of eight phases: $[5/4, 3/4, 1, 1, 1, 1, 1, 1]$ where each phase lasts one RT_{prop} . In the first phase, BBR sends 25% faster than B_{tLBw} to probe for additional bandwidth. In the second phase, BBR sends 25% slower than B_{tLBw} to drain any queues created in the last phase and to achieve fairness with other flows. In the remaining phases, BBR sends equal to B_{tLBw} ; the target operating point.

ProbeRTT. The goal of this state is to obtain a recent and accurate measurement of RT_{prop} . Since queue delay increases the measured RT_{prop} , ProbeRTT explicitly backs off from the network in order to drain any queues. This way, the min RT_{prop} filter can capture a RT_{prop} measurement without queues. ProbeRTT is entered if 10 seconds have elapsed since RT_{prop} was last updated, and lasts for 200 ms; long enough to overlap with other flow's ProbeRTT states such that queues are fully drained.

Recovery. This state is entered when data has been lost and exits once all outstanding data when Recovery was entered has been acknowledged. Upon entry, $cwnd$ is set to the amount of in-flight data and resets to $2 \times BDP$ upon exit.

Exponential Backoff. This state is entered upon a retransmission timeout indicating lost data due to no new acknowledgments for several RTTs. The lost segment is retransmitted with a doubled timeout time; exponentially back-

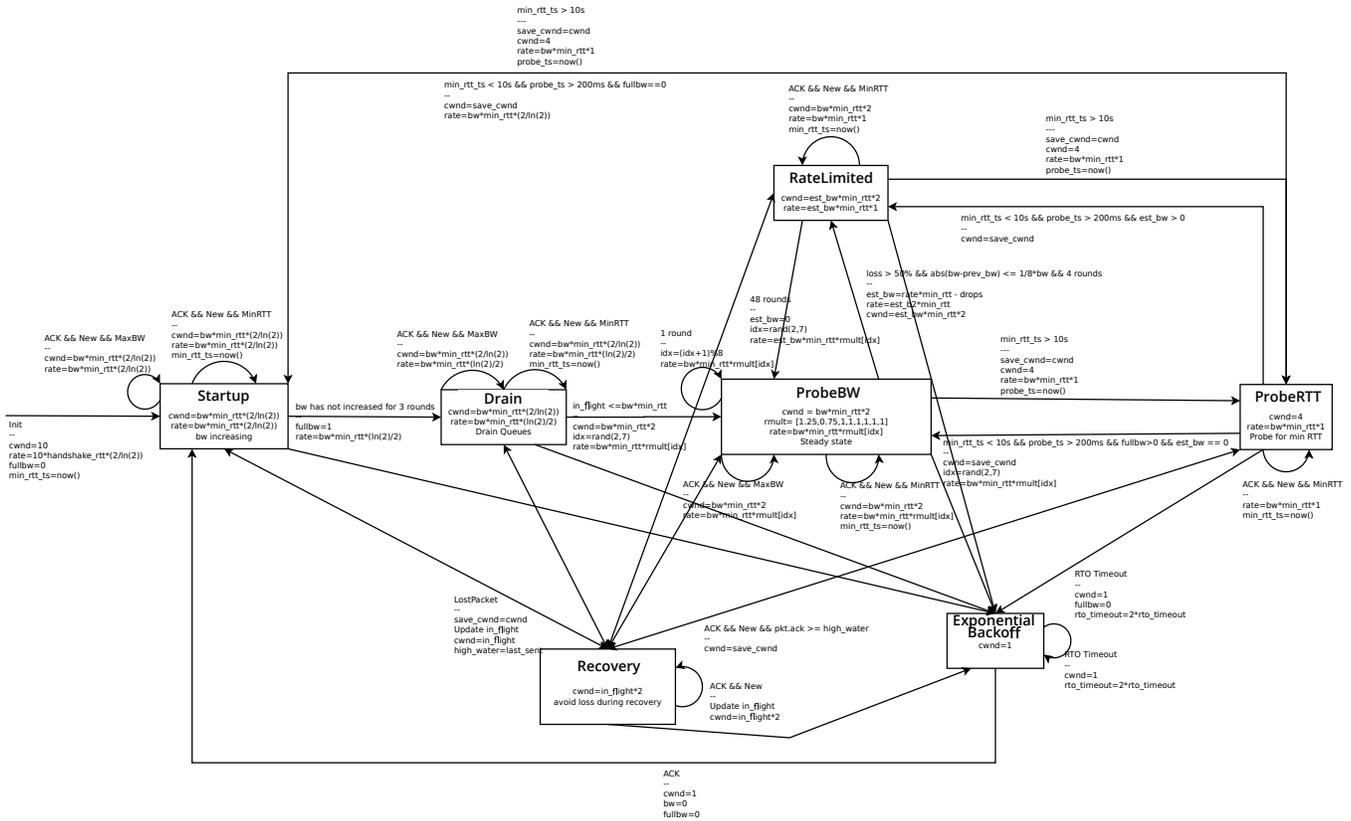


Figure 1: TCP BBR finite-state machine, see Table 1 for variable descriptions.

Table 1: Descriptions of variables unique to the BBR finite-state machine (left), and its events (right).

Variable	Description	Event	Description
<code>bw</code>	maximum measured bottleneck bandwidth.	ACK	recipient of an acknowledgment packet, representing the highest correct byte received.
<code>bw_est</code>	estimated token-bucket drain rate.	MaxBW	new maximum bottleneck bandwidth is observed.
<code>fullbw</code>	boolean indicating when pipe is filled.	MinRTT	new minimum RTT sample is observed.
<code>idx</code>	current index into <code>rmult</code> .	New	new acknowledgment received, acknowledging previously outstanding data.
<code>min_rtt</code>	minimum measured RTT.	LostPacket	TCP packet loss event (3 duplicate ACKs).
<code>min_rtt_ts</code>	timestamp <code>min_rtt</code> was measured.	RTO Timeout	outstanding data has not been acknowledged for many RTTs.
<code>probe_ts</code>	timestamp ProbeRTT was entered.		
<code>rate</code>	current pace data is sent.		
<code>rmult</code>	array containing the 8 <code> pacing_gain </code> phases.		

ing off from the network. Once an acknowledgment is received, the current model is discarded and Startup is entered.

Rate Limited. This state is entered when a token-bucket policer is detected on the network, as these can lead to high amounts of packet loss. This state is entered when the packet loss-to-delivered ratio is greater than 20%, but the throughput remains steady. BBR sets `Bt1Bw` to the estimate token bucket drain rate and sustains this for 48 round-trips.

3 Automated Attack Exploration in BBR

In order to systematically examine vulnerabilities of TCP BBR implementation for the Linux kernel TCP stack [8], we apply a TCP congestion control fuzzer, TCPWN. Below we

describe the attacker model and the changes we had to make to TCPWN in order to apply it to BBR.

3.1 Attacker Model

We focus on manipulation attacks in the implementation of BBR, where the attacker targets to mislead the sender’s congestion control about the current network condition. These attacks are conducted through maliciously crafted acknowledgment packets, which can result in either increasing or decreasing the throughput of the target flow(s), or in stalling the target TCP connection.

We support the following acknowledgement-based malicious actions: ACK duplication, ACK stepping (several acknowledgments are dropped and then several let through in a

cycle), ACK bursting (acknowledgments are sent in bursts), optimistic ACK (acknowledge highest byte, dropping duplicates), delayed ACK (delay acknowledgments for a fixed amount of time), limited ACK (prevent acknowledgment numbers from increasing), stretch ACK (forward only every n^{th} ACK), injecting off-path duplicate acknowledgments, injecting off-path offset acknowledgments, and injecting off-path incrementing acknowledgments. (Appendix B includes a visual representation of some actions for additional clarity.)

In order to achieve its goals, the attacker applies an *attack strategy*, which is defined as a sequence of acknowledgment-based malicious actions and the corresponding sender states when each action is conducted. We focus on TCP flows with bulk data transfers because they are widespread, and the effect of the conducted attacks is easy to measure.

We assume that the attacker is interested in causing BBR to send faster than usual, slower, or stall, and these attacks are meant to affect servers, clients, or the provider of a bottleneck link. In the case of sending faster, the goal of the attack can be to waste/exhaust bandwidth resources, worsening performance for all other clients of the server and/or shared bottleneck link. In the case of sending slower, the goal is to target individual connections for performance degradation, which could selectively cause a service provider's quality to be poor (e.g., low resolution video streaming) and/or make more bottleneck bandwidth available for other competing flows. In the case of stalling a flow, the goal is to disrupt communication between endpoints indefinitely, without causing an error from the transport layer to propagate to the application that is using it, effectively causing a denial of service.

3.2 Modifying TCPWN for BBR

We leverage TCPWN [29], a recent *open-source* platform designed to automatically find manipulation attacks in TCP congestion control implementations. We chose TCPWN because it does not require the source code of the congestion control implementation, and is designed specifically for TCP congestion control implementations.

At the core, TCPWN employs a network protocol fuzzer to find acknowledgment-based manipulation attacks against TCP congestion control implementations. Instead of applying random fuzzing, TCPWN guides the fuzzer using a model-guided technique, where the model is represented as a finite state machine (FSM) that captures the main functionality of several TCP congestion control algorithms.

For fuzzing an actual implementation of TCP congestion control in its native environment, TCPWN utilizes virtualization and proxy-based attack injection. To be effective, these attacks must be executed at the right time during execution, and therefore TCPWN monitors network packets exchanged to infer the current state of the sender in real-time.

While TCPWN is amenable to TCP congestion control algorithms, it assumes that the algorithm is a *loss-based* model

based on TCP New Reno. Thus, we can not directly apply it to BBR, as the models are substantially different. We leverage our own BBR FSM (see Figure 1) to feed it as an input to TCPWN to generate *abstract* attack strategies, each of which specifies a vulnerable path in the FSM that the attacker can exploit. Each transition on a vulnerable path dictates the network condition that the attacker needs to trigger to mount the desired attack.

While there are several ways to trigger the necessary network conditions, TCPWN takes the abstract strategies and converts them into *concrete* attack strategies consisting of *basic acknowledgement-packet-level actions* (e.g., send duplicate ACKs). During fuzzing, the attack injector applies these actions in particular states of the FSM. Although the generation of attack strategies is fully automated, TCPWN requires us to provide a manually crafted mapping between network conditions and basic actions because the mapping relies on domain knowledge about the underlying model (in our case, the BBR FSM).

Another change we had to make is changing the state inference algorithm. TCPWN needs to know what is the state of the sender in order to inject attacks in the states specified by the attack strategy. The state inference available in TCPWN cannot infer BBR's states because the algorithm expects the underlying model (*i.e.*, FSM) to be based on TCP New Reno. Hence, we develop a new state inference algorithm for BBR to infer the sender's state from network traffic alone (§ 3.3).

3.3 State Inference for BBR

We present a novel algorithm to infer the current state of the sender in real-time by passively observing network traffic. Our algorithm operates by computing flow metrics on each round-trip and comparing metrics across intervals to determine BBR's state. We compute metrics on each round-trip because BBR sustains a constant sending behavior for at least one round-trip.

When our algorithm starts, we begin a round-trip by recording the first data packet's sequence number and end when it is acknowledged. During round-trips, we collect flow metrics and compute average throughput, re-transmission count, number of data packets sent when the round completes. We then update the inferred BBR state on each new round by computing metrics across round-trips. For BBR, the most revealing metric about its current state is change in average throughput across round-trips. On each round, we compute the throughput *ratio* since the last round. For example, if the current and last rounds had average throughputs of 30 and 20 Mbit/sec respectively, then the *ratio* would equal 1.5.

We infer Startup if throughput has increase significantly since the last round-trip. Drain is primarily inferred if the current state is Startup and we notice throughput has not been increasing. ProbeBW is inferred if $1.4 > \text{ratio} > 0.6$, which allows ProbeBW to be inferred during phases 1 and 2 of gain

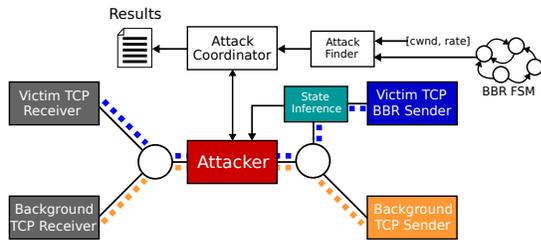


Figure 2: TCPWN testing environment.

cycling. We also infer ProbeBW when BBR transitions out of Drain indicated by a significant increase in throughput. We infer ProbeRTT when we observe only 4-5 data packets in the last round, resulting from $cwnd = 4$ packets to drain queues, and exit after 10 data packets have been sent. Recovery is inferred when re-transmitted segments have been observed and exits when the highest data sequence (when Recovery was entered) is acknowledged. Exponential-backoff is inferred when the estimated RTO has elapsed since the last data packet. Lastly, we infer RateLimited when more than 16 round-trips have passed, with small variance in average throughput.

We infer Exponential-backoff, Recovery and Drain without waiting for a round-trip to complete, as these can be entered at anytime during the flow. (Readers can refer to Appendix D for the algorithm’s pseudocode).

4 Experimental Results

In this section we describe and analyze our discovered attacks on BBR congestion control. We first describe the testing environment used. We then describe how we analyze and classified the attacks. Lastly, we discuss and illustrate the discovered attack classes in detail.

4.1 Experiment Setup

Environment. Our testing environment, shown in Figure 2, consists of four virtual machines running Ubuntu Linux 17.10 sharing a virtual dumbbell network topology. We limit the bottleneck bandwidth to 100Mbps/sec with a 500 packet queue and a 10 ms. end-to-end latency between either end of the topology. We configure the virtual network with reasonably low latency and high bandwidth, allowing us to isolate the impact of attack strategies on BBR in a “friendly” environment.

For each attack strategy, two TCP flows are instantiated, a victim and background flow, each transferring an identical 100MB file over HTTP. The attacker is located between either end of the topology. The victim flow uses a Linux TCP BBR sender, whose flow is injected with attacks where the background flow is not targeted. We use `tcpdump` to measure both flows’ performance, captured between the senders and the bottleneck. The victim flow is measured to understand the impact of the attack, and the background flow is measured to

investigate the impact of competing flows during attack, *e.g.*, to determine if there was collateral damage.

Attacks in the wild. For an attack to be effective in the wild, an attacker will need to be able to recognize that TCP flows are indeed using BBR and to be able to be on the path (all the attacks we found were from on-path attackers). An attacker can determine that TCP is using BBR for congestion control by examining the startup phase of a target connection. Specifically, during startup BBR doubles its send rate every round, even with loss, until the bottleneck bandwidth estimate is relatively stable. So an on-path attacker could drop a single packet early in the connection startup to see if the TCP sender exits exponential growth. If so, this is probably loss-based congestion control, *i.e.* not BBR. Being on path can be accomplished by compromising a router along the path or inserting ones self into the path by ARP spoofing or similar attacks. Cross-traffic could also impede the effectiveness of the attack. We conducted a few experiments where we varied the number of cross-traffic flows in an attempt to gauge the impact of such traffic on our attacks. In particular, we varied the number of background CUBIC flows from 1 to 32 while executing each of our attacks. We repeated each of these scenarios 10 times and found that our attacks continue to be effective even with this significant level of background traffic (see Table 3).

Attack strategies. We used TCPWN to execute 30,297 attack strategies for manipulating BBR’s sending rate. After each attack strategy is executed, it is classified into one of four categories: faster, slower, stalled (data transfer did not complete), and benign (no attack was detected). These categories represent the BBR’s respective *sending rate* performance. The attack categorization algorithm is the same as the one used in [29]. (For convenience, it is shown in Appendix A).

Attack analysis. After all 30,297 attack strategies were executed, 8,859 were flagged as potential attacks: 14 faster, 4,025 slower and 4,820 stalled. We initially focused on extracting the strategies that were most effective at manipulating BBR’s sending rate in each category. To identify which attack strategies were most effective at impacting BBR’s performance, we grouped attack strategies in each category (ignoring attack specific parameters) and sorted each by average sending rate.

While this allowed us to understand which exact strategies were most effective, the limitation with this method is that it does not reveal if any subset of actions is more effective over others. Take for instance the following attack strategy that hypothetically affects sending rate performance:

```
[ (StateA, Action1), (StateB, Action2), (StateC, Action3) ] .
```

While it is true that this attack strategy affects sending rate performance, the above method does not indicate if performing Action2 in StateB was *necessary* for causing it. To find which actions were most effective, we generated all possible attack action subset combinations for each category and sorted them by their occurrence in the original attack strategies. This allowed us to see which attack actions the

Table 2: Descriptions of discovered attack classes targeting Linux TCP BBR congestion control.

No.	Attack	Attacker	Description	Result
1	Optimistic ACK	On/off-path	Acknowledge the highest sent sequence number before it is received, hiding all losses. This causes an overestimated B_{tLEW} and for data to be send earlier/faster than otherwise.	Faster
2	Delayed ACK	On-path	Delay acknowledgments from reaching the sender from the receiver for a fixed amount of time, causing B_{tLEW} to be underestimated and data to be sent at a slower pace.	Slower
3	Repeated RTO	On-path	For the entire flow, prevent new data from being acknowledged causing a RTO. Optimistically acknowledge the lost segment causing Startup to be entered. This causes substantial amounts of time to be wasted (not sending data) during periods before each RTO.	Slower
4	RTO stall	On-path	Prevent new data from ever being acknowledged causing a RTO and exponential backoff to never exit. This causes the connection to stall as no new data will ever be sent.	Stalled
5	Sequence Number Desync	On/off-path	Acknowledge the highest sent sequence number before it is received, hiding all losses causing sequence numbers to desynchronize. Induce RTO causing exponential backoff to be entered. For each re-transmission, the receiver replies with a lower acknowledgement number than the sender expects, causing the lost segment to never be acknowledged and exponential backoff to never exit.	Stalled

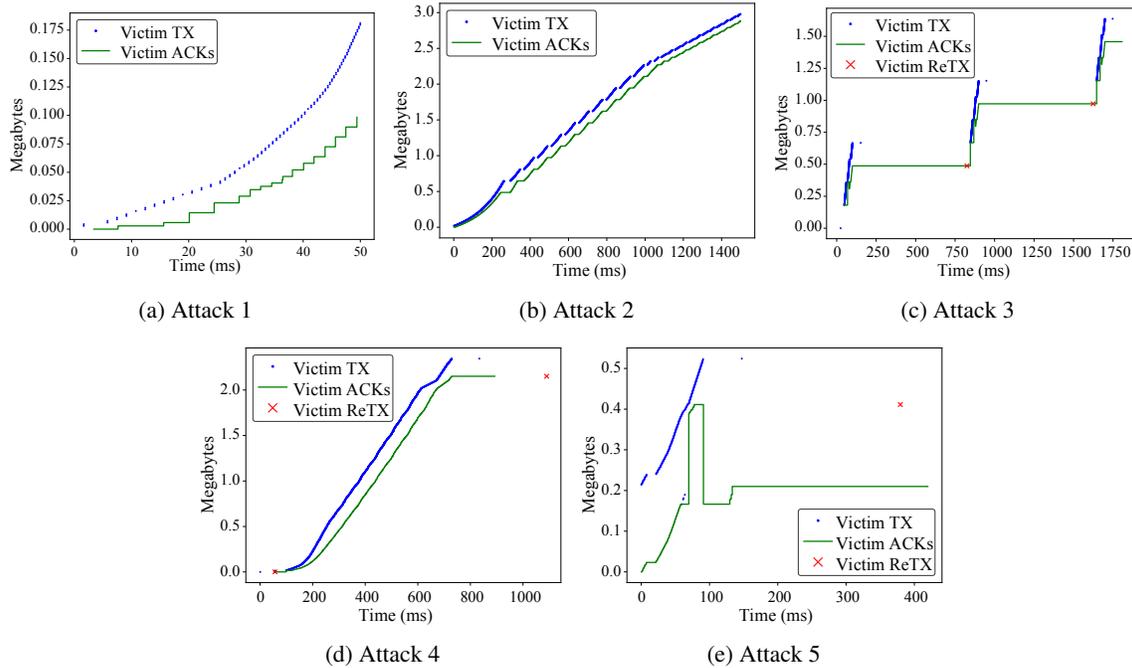


Figure 3: Time-sequence graphs illustrate how each attack manipulates acknowledgments to achieve a faster, slower or stalled sending rate. The blue lines represent data being sent by TCP BBR (victim) and the green represents acknowledgments being received from the attacker.

Table 3: Avg. throughput (in **Mbps**) of target BBR flow during attacks with varying numbers of CUBIC flows as cross-traffic

Attack	Background Flows					
	1	2	4	8	16	32
None	51.7	57.9	21.0	14.3	6.7	3.9
Attack 1	287.4	236.3	102.4	78.7	85.5	76.8
Attack 2	3.6	4.2	4.7	4.0	4.8	14.2
Attack 3	3.1	0.6	4.7	1.1	0.9	0.3
Attack 4	0.7	1.5	0.1	0.06	0.02	0.01
Attack 5	7.0	0.7	0.06	0.03	0.02	0.01

above method were most effective in each category.

For attack strategies causing faster send rates, the optimistic acknowledgment attack appeared in 100% of the strategies. For attack strategies causing slower send rates, executing the delayed acknowledgment attack in ProbeBW appeared in 53%

of the strategies, while attacks causing RTOs and quickly acknowledging data in exponential backoff appeared in 47% of the strategies. As for the attack strategies causing a stalled connection, attacks causing RTOs and preventing data from being acknowledged in exponential backoff appeared in 89% of the strategies. Finally, attacks that optimistically acknowledged lost data appeared in 11% of the strategies (see Table 2).

We use time/sequence graphs (TSGs) in Figure 3 and 4 to understand why these attacks affect BBR’s sending rate on a per-ACK basis. In each TSG, the x -axis is time and the y -axis is sent/acknowledged bytes from the BBR sender’s (victim’s) point-of-view. We use TSGs to also understand how BBR flows targeted by these attacks compare to benign flows (flows without attacks taking place) and how they affect background flows that share the network concurrently. Figure 5 shows the

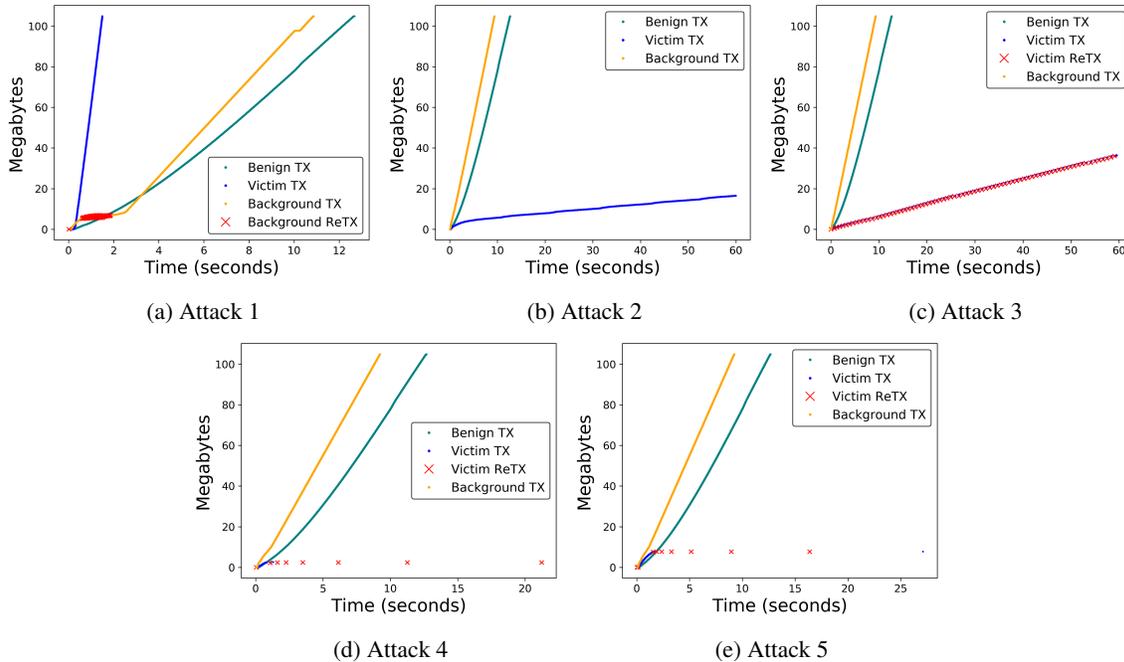


Figure 4: To understand how the victim flow for each attack compares to the background flow, these time-sequence graphs illustrate each attack is carried out during the entirety of a flow (100 MB transfer). For testing, we limit flows to 60 seconds. The orange and blue lines represent the victim and background flow’s cumulative data transfer over time (executed concurrently). The green line represents the benign flow with no attacks taking place (executed as a separate experiment). In attack 1, the background flow is unable to obtain bandwidth until the victim flow completes. In attack 2, 3, 4 and 5, the background flow obtains greater bandwidth due to the victim sending slower or stalling.

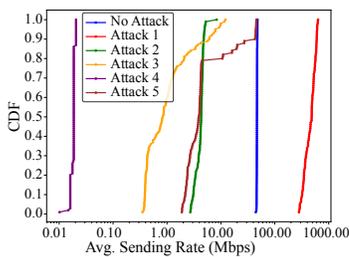


Figure 5: Each CDF represents the distribution of the avg. sending rate for each attack class executed 100 times. The victim BBR flow shares the network with an identical benign background CUBIC flow.

impact of executing each attack 100 times, using a CDF of the average send rate for the victim flow in each execution. Note that most curves are nearly vertical, indicating that each attack had a high probability of affecting the sending rate.

4.2 Discovered Attacks on BBR

Attack 1 – Optimistic Acknowledgments. A faster sending rate is caused by optimistically acknowledging *only* new data sequences sent by the sender, without sending duplicates, effectively causing BBR to overestimate the bottleneck

Table 4: The optimistic acknowledgment attack causes BBR to increase its sending rate by 25% every 8 round-trips. In this example, this attack effectively cuts the perceived RTT of 20 ms in half.

Time (ms)	Mbit/sec	Time (ms)	Mbit/sec
0	6.0	560	28.6
80	7.5	640	35.7
160	9.4	720	44.7
240	11.7	800	55.9
320	14.6	880	69.8
400	18.3	960	87.3
480	22.8	1040	109.1

bandwidth. The attacker records the highest observed data sequence number sent by the sender and modifies acknowledgment numbers from the receiver such that they acknowledge the highest sequence. If the modified acknowledgment would send a duplicate acknowledgment, then it is dropped. This results in packet loss (indicated by duplicate acknowledgments) to be hidden. An off-path attacker who is able to predict the sender’s sequence number and the receiver’s acknowledgment rate would be able to achieve the same impact on a victim flow by maliciously injecting acknowledgments such that they acknowledge new data sent by the sender.

A faster sending rate (see Figure 3a) is a byproduct of how BBR aggressively probes for additional bandwidth by sending 25% faster than $B_{t.lBw}$ for $1/8$ RTTs. This attack causes the acknowledgment rate to reflect the increased sending rate.

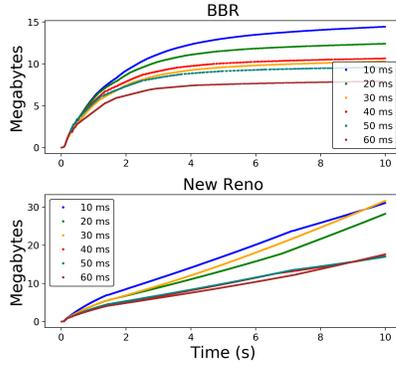


Figure 6: To understand how delayed ACKs affect the sending rate of BBR and New Reno, this figure illustrates several time-sequence graphs, each with a distinct delay time. Each line represents the cumulative amount of data sent during the connection. Due to different underlying techniques, delayed acknowledgments cause BBR’s sending rate to decay over time, where New Reno maintains a rate.

This is reflected in the delivery rate samples (see Algorithm 1) causing the estimated bottleneck bandwidth to increase by 25% as well. BBR maintains the increased sending rate for the next 8 round-trips until it sends 25% faster again on the next probing phase. Surprisingly, we discovered that this attack alone does not cause the sending rate to increase. Even though this attack caused the acknowledgment rate to increase (due to a shortened `ack_elapsed`), delivery rate samples are capped by the sending rate. It is not until BBR probes for bandwidth when this attack becomes effective, meaning it is self-induced. This attack also halves RT_{prop} because data is acknowledged sooner causing BBR to cycle through the 8 gain phases twice as fast. In our testing environment, this attack caused BBR to increase its sending rate from 6 Mbit/sec [11] to 800 Mbit/sec in less than 2 seconds! Table 4 shows how BBR’s sending rate exponentially grows in our testing environment.

Attack 2 – Delayed Acknowledgments. A slower sending rate (see Figure 3b) is caused by delaying acknowledgment packets from reaching the sender for a fixed amount of time, causing the bottleneck bandwidth to be underestimated. Interestingly, this attack caused BBR’s sending rate to grow inversely to the optimistic acknowledgments attack. Figure 6 shows how BBR’s sending rate grows in $O(\frac{1}{n \cdot \ln(\text{delay})}) = O(\frac{1}{n})$ time (derivative of $\log_{\text{delay}} n$). In general, longer delays (that do not cause the re-transmission timer to expire) caused BBR to decrease its sending rate quicker. The amount of data sent over time is not to be confused with the rate of change of its sending rate, hence the derivation.

This attack causes BBR’s sending rate to sequentially decrease over time because when this attack first starts, the sender experiences an initial delay in acknowledgments (equal to the however long ACKs are delayed for) after which acknowledgments arrive at their natural rate. This causes the sender to stop sending new data until the ACKs after the de-

lay arrive. Since the attacker stops sending data, this causes the sender to experience another delay in acknowledgment packets after one RTT. This creates a pattern where the sender experiences delays in acknowledgments every RTT, meaning regardless when delivery rate samples are taken, the plateaus in acknowledgments cause the delivery rate samples to always be less than the current Bt_{lBw} . This implies BBR will never increase its sending rate because as older Bt_{lBw} estimates expire after 10 RTTs, it is replaced with the decreased delivery rate samples due to this attack.

Surprisingly, BBR’s probing phase does not mitigate the effect this attack has on the sending rate. One would think that when BBR probes for bandwidth, the delivery rate samples computed during probing would be large enough to surpass the decreased delivery rate samples. This would be true, however because BBR drains queues immediately following probing, the delivery rate samples take during probing are still less than the current Bt_{lBw} . If BBR sent at a steady pace for at least 1 RTT in between probing and draining, then this attack would not result in its sending rate to decay.

Figure 6 shows how this attack is less effective on congestion control that uses AIMD such as TCP New Reno. Although this attack is still effective on New Reno, its sending rate maintains a constant decreased rate rather than decaying. In essence, New Reno observes this attack as an increase in RTT, resulting in slower growth and lower overall throughput, while BBR’s estimation mechanisms are unable to cope with such large RTT increases, resulting in incorrect estimates and more serious and long lasting issues.

It should be noted that this attack does not require TCP header information to be modified to be effective, as packets are only delayed. This implies QUIC [25], Google’s experimental transport layer protocol that uses encrypted headers, using BBR can be targeted by this attack.

Attack 3 – Repeated RTO. A slower sending rate is caused by an attacker who allows small amounts of data to be sent in between repeated re-transmission timeouts. This causes a slower sending rate because the sender does not send any new data until the RTO expires, and this cycle repeats throughout the duration of the transmission.

This attack begins by causing the sender to re-transmission timeout, which is achieved by preventing new data from being acknowledged. There are four acknowledgment-based manipulation actions that were found to cause this: dropping, limiting, stretching and delaying acknowledgments. Dropping acknowledgments simply consists of preventing ACKs from being delivered to the sender. By limiting acknowledgments, acknowledgment numbers are such that they equal $\min(\text{ack}, \text{limit})$. Stretching acknowledgments consists of forwarding only every n^{th} acknowledgment to the receiver. Lastly, delaying acknowledgments (also used in attack 2) consists of delaying acknowledgments from reaching the sender.

After the re-transmission timeout is achieved from one of the above methods, the sender enters exponential backoff

and the attacker optimistically acknowledges some data and repeats the process. The purpose here is to quickly acknowledge the lost segment in order to cause BBR enter Startup. When BBR transitions into Startup, its network path model (Bt_{LBw} and RT_{prop}) is discarded, meaning the model must be rediscovered after each RTO. This attack prevents BBR from obtaining an opportunity to send data anywhere near the optimal operating point, resulting in decreased throughput. Instead, data is sent in bursts with lengthy idling in between.

Figure 3c illustrates these bursts and how the connection idles until the timeout. At 0 ms, BBR is in Startup, and the attacker begins to drop, limit, stretch or delay ACKs. As a result, the sender stops sending data because in_flight has reached $cwnd$. When the timeout occurs around 800 ms, the attacker optimistically acknowledges the lost segment, causing Startup to be reentered, after which the attack repeats. In Figure 4c, the victim flow can be seen experiencing timeouts throughout the entire flow (indicated by the red x-markers), allowing the background flow to obtain more bandwidth.

Interestingly, the work in [29] reported a similar attack resulted in a faster sending rate in several cases. They note how the idle periods in between timeouts is outweighed by repeatedly entering slow start (where $cwnd$ doubles on each ACK). This was not the case in our work, however, most likely due to how we induce timeouts the very instant BBR enters Startup, resulting in less time for the sending rate to double.

Attack 4 – RTO Stall. In this attack, a stalled connection is caused by an attacker who causes BBR to enter exponential-backoff, and prevents it from ever exiting. The attack begins by causing the sender to timeout by dropping, limiting, stretching or delaying acknowledgments. After the timeout and when exponential backoff is entered, the attacker prevents any new data from being acknowledged, by limiting or dropping acknowledgments. This causes BBR to permanently remain in exponential backoff because the re-transmitted segment will never be acknowledged. Additionally, no new data will ever be sent because in_flight reached $cwnd$, effectively stalling the connection. The lost segment will be re-transmitted 15 times (Linux default) with a doubled timeout time in between each re-transmit. On the 16th re-transmission, the TCP connection would be torn down by the sender (at least 15 minutes, 24 seconds from the first re-transmission).

In Figure 3d, the connection stalls around 700 ms after only sending about 2.5 MB. In Figure 4d, the background flow is able to obtain greater bandwidth made available by the victim stalling. It is important to note that this attack is highly flexible as it can be applied at any time during a connection and is not limited to a specific state or time.

Attack 5 – Sequence Number Desync. In this attack, a stalled connection is caused by an attacker who optimistically acknowledges lost data causing sequence numbers between the sender and receiver to de-synchronize. This attack works by acknowledging a lost segment (that was not actually delivered to the receiver). The primary reason why this attack

causes a stalled connection is because the sender is unable to re-transmit the lost segment because it was removed from the “re-transmission queue”. The TCP write queue retains segments until they have been acknowledged. As segments are acknowledged, they are discarded from the queue in order to free memory.

Next, the sender transmits the next data segments, which will be delivered out-of-order from the receiver’s point of view, meaning the receiver will respond by acknowledging the highest correct data segment received so far. Since the out-of-order segment the sender sends cannot be acknowledged, it will keep being re-transmitted eventually causing three duplicate acknowledgments to be sent by the receiver. When this occurs, the sender will try to re-transmit the lost segment but cannot because it has been removed from the re-transmission queue.

In Figure 3e, the connection stalls because while the receiver sends duplicate acknowledgments (around the 0.2 MB mark), the sender keeps re-transmitting the same segment (around the 0.4 MB mark) due to RTOs. Since the receiver cannot receive that segment because it is out-of-order, it cannot acknowledge it, causing a stalled connection. In Figure 4e, the background flow can be seen slight increasing its sending rate because when the victim’s connection stalled, more bandwidth is made available, allowing the background flow to send faster. This attack was discovered in [29] for TCP New Reno which also resulted in a stalled connection.

Although this attack is most effective from an on-path attacker, this attack could be achieved by an off-path attacker who successfully learns the victim flow’s sequence number state. If an off-path attacker successfully crafts and injects an acknowledgment packet acknowledging a lost segment, then a stalled connection would result.

4.3 Ineffective Attacks Against BBR

We now describe how some previously known attacks against congestion control were ineffective against BBR. (Readers can refer to Appendix C for a visual representation for these attacks.)

Acknowledgment Bursts. In this attack, the attacker accumulates n acknowledgment packets from the receiver before forwarding them to the sender in a single burst. In [29], this attack caused New Reno to send data in bursts because TCP is ACK-clocked meaning its sending behavior closely mimics the acknowledgment behavior. In BBR, acknowledgment bursts do not cause data to be sent in bursts because even though the delivery rate samples computed for the first $n - 1$ ACKs in the burst are deflated, the delivery rate sample for the n^{th} (last) ACK in the burst is no different than without an attack. Since the delivery rate samples for the first $n - 1$ ACKs are always less than the n^{th} (last) ACK, the ACKs arrive at the same time and larger delivery rate samples take precedence, this attack does not impact BBR’s Bt_{LBw} estimate.

Acknowledgment Division. In this attack, a single ACK acknowledging m bytes is divided into n valid ACKs each acknowledging roughly m/n bytes. In [36], this attack caused `cwnd` to grow n times as fast because for each ACK, `cwnd` increased by one segment. Depending on how the attacker injected the divided ACKs, this attack had no effect on BBR’s sending rate for different reasons. If the attacker sent the divided ACKs at the same time as the valid ACK (the ACK being divided), then the attack would not be effective for the exact same reason as to why the acknowledgment burst attack did not affect BBR’s sending rate. If the attacker sent the divided ACKs at the same time as the last ACK (the ACK before the one that is being divided), then the ACK rate would be clamped by BBR’s current sending rate. If the attacker performed this during BBR’s probing phase, then it would be identical to attack 1 where the sender’s sending rate is increased. If the attacker evenly spaced each divided ACK, then the ACK rate of the divided ACKs would be no different than the ACK rate without the attack.

Duplicate Acknowledgments. In this attack, n duplicate acknowledgments are injected for every acknowledgment packet from the receiver. In congestion control schemes such as CUBIC and New Reno that use packet loss to detect congestion, when ≥ 3 duplicate acknowledgments were injected, Fast Recovery was entered to re-transmit the lost segment. This caused a decreased sending rate because upon entering Fast Recovery, the congestion window is halved causing data to be sent at a slower rate. Although BBR is not loss-based, it still includes a mechanism for dealing with packet loss (by detecting duplicate acknowledgments) by entering a Recovery state. The reason this attack is not effective against BBR is because BBR does not backoff from the network upon packet loss. Instead, BBR sets `cwnd` to `in_flight` and re-transmits the lost segment until all outstanding data when Recovery was entered is acknowledged.

5 Defenses

Most of our attacks rely on being able to modify acknowledgment information in TCP packets. The best defense against these attacks is to encrypt or authenticate this information. QUIC, a new transport protocol initially developed by Google but currently being standardized by the IETF, takes this approach. Unfortunately, adding this kind of authentication to TCP is impractical due to backwards compatibility issues. Similarly, prior work [39] has suggested adding a nonce to TCP acknowledgments to prevent optimistic ACK attacks. This suffers from similar backwards compatibility issues. Finally, some attacks, like the Delayed ACK attack, require only the ability to delay/reorder packets and appear to be inherent in trying to infer model parameters from delivered packets.

6 Related Work

Congestion Control Attacks. The work in [36] demonstrates how a misbehaving receiver can undermine congestion control making senders send data at a faster pace without compromising reliability. It is shown how TCP is susceptible to divided, duplicate and optimistic acknowledgment attacks.

Much work has gone into *off-path* attackers who have write-only access to a flow. Sequence numbers can be predicted [5, 7, 18, 23, 33–35] to inject malicious content into a victim’s connection. The work in [35] shows how sequence numbers can be leaked to unprivileged, on-device malware to coordinate with an off-path attacker, yielding connection hijacking in under one second. The work in [5] aims for better initial sequence number generation to make it more difficult for off-path attackers to succeed.

Protocol Fuzzing. Program analysis by automatically generating inputs has long been used to test for security, robustness and reliability. Instead of generating random inputs, the work in [22] takes an approach by generating relevant tests tailored to all possible source code paths. Similar approaches have been used for network protocol analysis. MAX [32] discovers attacks in network protocols however requires source code to be annotated where vulnerabilities are likely to exist, yielding thorough manual analysis. This motivated model-guided testing [19, 20, 29] where a protocol’s state machine is used to discover relevant attacks which has been applied to a variety of protocols. KiF [1], SNOOZE [4] and SNAKE [28] all take model-guided approaches to discover relevant and effective attack strategies in network protocols. TCPWN [29] takes a model-guided approach for discovering acknowledgement-based manipulation attacks in TCP congestion control implementations. As discussed in Section 3, TCPWN can not be directly applied to BBR.

7 Conclusion

We identified 5 classes of attacks from on-path attackers that caused BBR to send data at a high, slow and stalled rates. We found that due to how BBR multiplicatively probes for bandwidth, an attacker who optimistically acknowledges data caused BBR to increase its sending rate by 13x in under 1 second. We showed that the combination of gain cycling and delayed acknowledgments by an attacker caused BBR to sequentially decrease its sending rate. We also showed that an attacker that prevented new data from being acknowledged caused re-transmission timeouts and for BBR to reset and rediscover the network path model each time. We also identified two attacks that stall data transmission: an attacker who prevents new data from being acknowledged and an attacker that optimistically acknowledges lost data causing sequence numbers to desynchronize. Finally, we show how the burst, divide and duplicate acknowledgment attacks against prior congestion control schemes are not effective against BBR.

References

- [1] Humberto J. Abdelnur, Radu State, and Olivier Festor. KiF: A Stateful SIP Fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 47–56, New York, NY, USA, 2007. ACM.
- [2] Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the Performance of TCP Pacing. *Proceedings - IEEE INFOCOM*, 01 2000.
- [3] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: Emulation and Experiment. *SIGCOMM Comput. Commun. Rev.*, 25(4):185–195, October 1995.
- [4] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. 2006.
- [5] S. Bellovin. Defending Against Sequence Number Attacks. <https://tools.ietf.org/html/rfc1948>, 1996.
- [6] Lawrence S. Brakmo, Sean W. O’malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *In SIGCOMM*, 1994.
- [7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP Exploits: Global Rate Limit Considered Dangerous. In *Proceedings of the 25th USENIX Conference on Security Symposium, SEC’16*, pages 209–225, Berkeley, CA, USA, 2016. USENIX Association.
- [8] N. Cardwell, J. Priyaranjan, E. Dumazet, K. Yang, D. Miller, and Y. Seung. Linux TCP BBR. https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/tree/net/ipv4/tcp_bbr.c, 2018.
- [9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.
- [10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control. <https://www.ietf.org/proceedings/97/slides/slides-97-iccr-g-bbr-congestion-control-02.pdf>, November 2016.
- [11] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control: An Update. <https://www.ietf.org/proceedings/98/slides/slides-98-iccr-g-an-update-on-bbr-congestion-control-00.pdf>, March 2017.
- [12] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, and Van Jacobson. BBR Congestion Control: IETF 100 Update: BBR in shallow buffers. <https://datatracker.ietf.org/meeting/100/materials/slides-100-iccr-g-a-quick-bbr-update-bbr-in-shallow-buffers>, November 2017.
- [13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, and Van Jacobson. BBR Congestion Control: IETF 99 Update. <https://www.ietf.org/proceedings/99/slides/slides-99-iccr-g-iccr-g-presentation-2-00.pdf>, July 2017.
- [14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, and Van Jacobson. BBR Congestion Control Work at Google: IETF 101 Update. <https://datatracker.ietf.org/meeting/101/materials/slides-101-iccr-g-an-update-on-bbr-work-at-google-00>, March 2018.
- [15] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, Kevin Yang, Matt Mathis, and Van Jacobson. BBR Congestion Control Work at Google: IETF 101 Update. <https://datatracker.ietf.org/meeting/102/materials/slides-102-iccr-g-an-update-on-bbr-work-at-google-00>, July 2018.
- [16] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control. <https://tools.ietf.org/id/draft-cardwell-iccr-g-bbr-congestion-control-00.html>, 2017.
- [17] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. Delivery Rate Estimation. <https://tools.ietf.org/html/draft-cheng-iccr-g-delivery-rate-estimation-00>, 2018.
- [18] Weiteng Chen and Zhiyun Qian. Off-Path TCP Exploit: How Wireless Routers Can Jeopardize Your Secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1581–1598, Baltimore, MD, 2018. USENIX Association.
- [19] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *USENIX Conference on Security*, 2011.

- [20] Joeri de Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., 2015. USENIX Association.
- [21] Defense Advanced Research Projects Agency. Transmission Control Protocol. <https://tools.ietf.org/html/rfc793>, 1981.
- [22] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering, ICSE '81*, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.
- [23] Yossi Gilad and Amir Herzberg. Off-path Attacking the Web. In *Proceedings of the 6th USENIX Conference on Offensive Technologies, WOOT'12*, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.
- [24] University of Southern California Information Sciences Institute. Transmission Control Protocol. <https://tools.ietf.org/html/rfc793>, 1981.
- [25] J. Iyengar, Ed and Fastly and M. Thomas, Ed and Mozilla. QUIC: A UDP-Based Multiplexed and Secure Transport. <https://tools.ietf.org/html/draft-ietf-quic-transport-18>, 2019.
- [26] Van Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.
- [27] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *SIGCOMM Comput. Commun. Rev.*, 19(5):56–71, October 1989.
- [28] S. Jero, H. Lee, and C. Nita-Rotaru. Leveraging State Information for Automated Attack Discovery in Transport Protocol Implementations. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.
- [29] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach. In *Proc. of Network & Distributed System Security Symposium (NDSS)*, 2018.
- [30] Laurent Joncheray. A simple active attack against TCP. In *USENIX Security Symposium*, 1995.
- [31] L Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. 01 1979.
- [32] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govidan, and Madanlal Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, pages 26–37, 2011.
- [33] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software, 1985.
- [34] Z. Qian and Z. M. Mao. Off-path TCP Sequence Number Inference Attack - How Firewall Middleboxes Reduce Security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361, May 2012.
- [35] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *ACM Conference on Computer and Communications Security*, 2012.
- [36] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM SIGCOMM Computer Communication Review*, 29(5), 1999.
- [37] V. Jacobsen and LBL and R. Braden and ISI. TCP Extensions for Long-Delay Paths. <https://tools.ietf.org/html/rfc1072>, October 1988.
- [38] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006.
- [39] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-way Traffic. *SIGCOMM Comput. Commun. Rev.*, 21(4):133–147, August 1991.

A Attack Strategy Categorization

Attack strategy categorization algorithm is shown in Algorithm [A1](#).

B Illustrations of Malicious Actions Used by our Attacks Strategies against BBR

See Figure [B1](#).

C Illustrations of Ineffective Attacks against BBR

See Figure [C2](#).

D BBR State Inference Algorithm

For completeness, we present a pseudocode of the BBR state inference algorithm in Algorithm [D2](#).

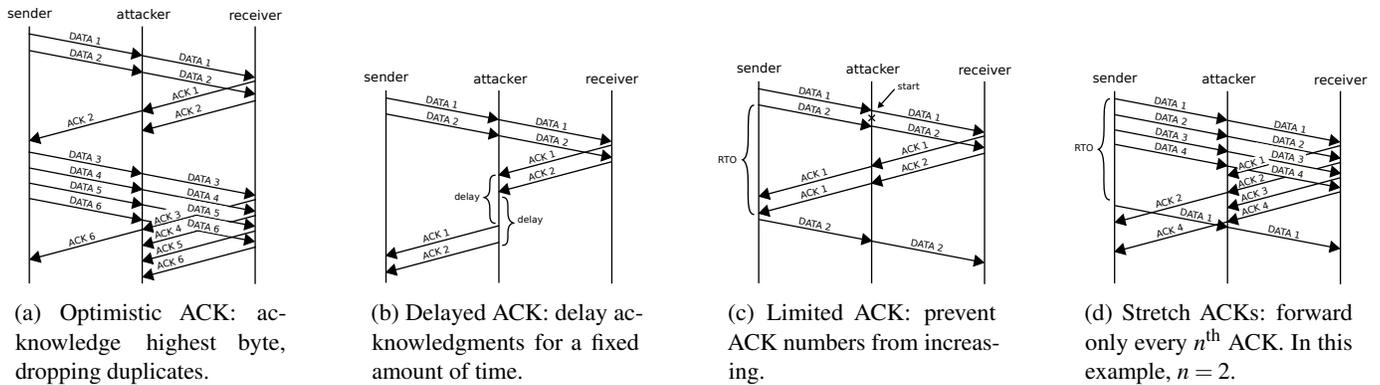


Figure B1: Time lines of acknowledgment-based manipulation actions used in our attack strategies.

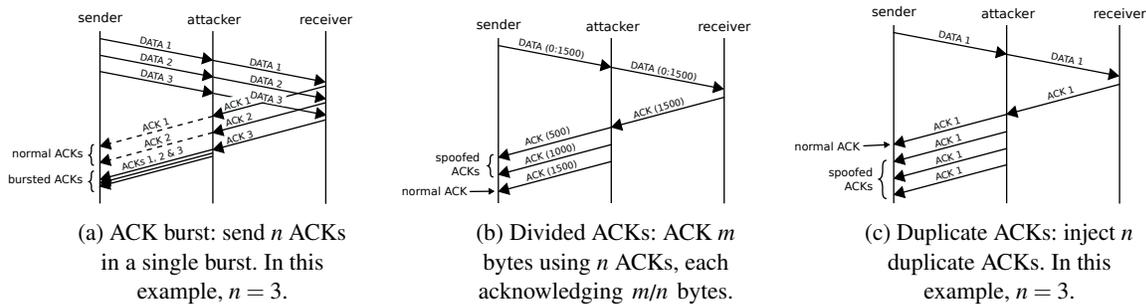


Figure C2: Time lines of acknowledgment-based manipulation actions that were previously known to be effective against TCP congestion control, but were ineffective against BBR.

Algorithm A1 Attack strategy categorization algorithm: 20 benign experiments are first executed to obtain a baseline average and standard deviation. Since each strategy transfers the same 100MB file, an strategy is categorized as a function of its total transfer time and the baseline average and standard deviation transfer time.

Input: Strategy execution metrics

Output: The category of the strategy

```

1: function CATEGORIZEATTACKSTRATEGY(s)
2:   if  $s.Time > (s.TimeAvg + 2 * TimeStddev)$  then
3:     return SLOWER
4:   else if  $s.Time < (TimeAvg + 2 * TimeStddev)$  then
5:     if  $s.SentData \geq (0.7 * 100MB)$  then
6:       return FASTER
7:     else
8:       return STALLED
9:   end if
10:  else
11:    return BENIGN
12:  end if
13: end function

```

E Background on Congestion Control

E.1 Congestion Control Overview

Congestion control determines whether to send a segment of data based on information inferred about a network path between endpoints. A primary goal is to saturate the bottleneck link (maintaining high utilization) while avoiding congestion collapse (due to sending faster than the bottleneck link can support). The bottleneck link is saturated when the total amount of data in-flight equals the path's bandwidth-delay product (BDP) which represents the maximum amount of in-flight data the network [31] can process without dropping packets. A path's BDP is dynamic and typically computed as the product of the bottleneck link's maximum bandwidth and the path's round-trip time without queue delay [37]. Congestion control is also tasked with avoiding congestion collapse and achieving fairness with other flows sharing the network. Accomplishing these goals is challenging because networks are unpredictable: links vary in bandwidth capacity and are shared anywhere between few to millions of hosts such as the global Internet. While several congestion control algorithms have been developed, most adhere to the same basic principles first described by Jacobson in 1988 [26]. Below we describe the main goals of a congestion control algorithm.

Discovering the target sending rate. The target sending

Algorithm D2 BBR state inference algorithm

```
function ONNEWBBRPACKET(packet)
    newInt = CHECKINTERVAL(packet) /* a round-trip has passed */
    if dataPackets > 0 and RTPASSEDFORDATAPACKET() then
        priorState = currentState
        currentState = EXPONENTIAL_BACKOFF
        COMPUTEINTERVALMETRICS()
    else if retransmissions > 0 and not currentState == RECOVERY then
        priorState = currentState
        currentState = RECOVERY
        /* leave recovery when high water is ACK'd */
        highWater = highestDataSequence
    else if currentState == RECOVERY then
        ratio = THROUGHPUTRATIOSINCELASTROUNDTRIP()
        if totalAcked ≥ highWater then
            currentState = priorState /* return to previous state */
            SETRETRANSMISSIONCOUNT(0)
            else if newInt and priorState == STARTUP and 0.7 > ratio > 0.1 then
                priorState = DRAIN
            end if
        else if currentState == PROBERTT and dataPackets > 10 then
            currentState = priorState
        else if newInt then
            currentState = UPDATEBBRNEWINTERVAL()
        else if dataPackets ≥ 10 then
            /* check for drain on frequent basis */
            ratio = THROUGHPUTRATIOSINCELASTROUNDTRIP()
            if currentState == STARTUP and 0.7 > ratio > 0.1 then
                priorState = currentState
                return DRAIN
            end if
        end if
    end if
end function

function UPDATEBBRNEWINTERVAL()
    ratio = THROUGHPUTRATIOSINCELASTROUNDTRIP()
    dataPackets = GETDATAPACKETCOUNT()
    nonIncrease = NONINCREASEINTERVALCOUNT()
    if 6 > dataPackets > 3 then /* small amount of data in-flight */
        return PROBERTT
    else if currentState == DRAIN and ratio > 1.4 then
        priorState = currentState
        return PROBEBW
    else if not currentState == PROBEBW and ratio > 1.4 then
        priorState = currentState
        return STARTUP
    else if currentState == STARTUP and 0.7 > ratio > 0.1 then
        priorState = currentState
        return DRAIN
    else if currentState == STARTUP and nonIncrease > 10 then
        priorState = currentState
        return DRAIN
    else if intervalCount > 16 and throughputVariance < 100 then
        priorState = currentState
        return RATE_LIMIT
    else if not currentState == STARTUP and 1.4 > ratio > 0.6 then
        priorState = currentState
        return PROBEBW
    else
        return currentState
    end if
end function

function CHECKINTERVAL(packet)
    if not inInterval and ISDATAPACKET(packet) then
        CLEARINTERVALDATA()
        ackMark = packet.sequenceNumber
        inInterval = true
    end if
    if inInterval and packet.ackNumber ≥ ackMark then
        inInterval = false
        COMPUTEINTERVALMETRICS()
        return true
    end if
    return false
end function
```

rate is one that achieves high throughput and avoids congestion. The sending rate is dictated by a per-connection variable known as the congestion window $cwnd$ which governs the maximum amount of unacknowledged data allowed in-transit. When a connection first starts, congestion control performs *slow start* to quickly discover the available bandwidth of the link. Afterwards, *congestion avoidance* is performed whereby data is sent conservatively while slowly probing the network for available bandwidth.

Inferring congestion. Congestion control must use signals from the network to infer congestion as an indicator to “back off”, or reduce its load on the network. The most popular paradigms have been loss-based congestion control and delay-based congestion control. Loss-based congestion control has dominated the Internet since its creation and uses packet loss as signal of congestion. Packet loss occurs when switch buffers along a network path fill to capacity and are left with no choice but to discard, *i.e.*, drop, incoming packets. Delay-based congestion control compares predicted and actual round-trip time (RTT) samples to signal congestion. These signals, be they packet loss or RTT, govern the rate at which data is sent into the network.

Achieving fairness with competing flows. Since networks today are shared by several end-hosts, congestion control aims to share the limited resources of a bottleneck link evenly across all flows. Coexisting congestion control algorithms may not be fair to each other due to differing probing and backoff mechanisms. For example, delay and loss-based congestion control do not operate well together. Delay-based congestion control has been shown to reduce its congestion window much earlier than loss-based [3], resulting in unfair bandwidth allocation. Because of this, delay-based congestion control is not commonly used in today’s networks.

E.2 Congestion Signal

Loss-based congestion control uses two signals to detect packet loss: re-transmission timeouts (RTOs) and duplicate acknowledgment packets. Delay-based congestion control leverages changes in RTT samples to infer congestion. All methods leverage feedback from the receiver (*i.e.* acknowledgment packets) to detect congestion.

Re-transmission timeout. RTOs ensure data delivery when there is no feedback from the receiver. Each time a data segment is sent, a timer starts and expires if the segment has not been acknowledged after a certain amount of time (usually several RTTs). Each time the timer expires, the data segment is re-transmitted and the timer restarts with a doubled timeout time. The initial timeout time is typically a function of RTT samples, gathered over the duration of a connection. This event can indicate congestion so severe that acknowledgments cannot be delivered in a sufficient amount of time.

Duplicate acknowledgments. When an out-of-order data

segment is received, a receiver will ignore its payload and reply acknowledging the last correctly received data byte. If a data segment becomes lost in-transit, then the following in-transit segments will be received out-of-order, causing the receiver to send several duplicate acknowledgments. When a sender receives multiple (*e.g.*, three) duplicate acknowledgments in a row, congestion is inferred. This event signifies less severe network congestion because despite the loss of a data packet, acknowledgments are still able to be received by the sender.

Delayed acknowledgments. Delay-based congestion control, used by TCP Vegas [6], FAST TCP [38] and CARD [27], takes a *proactive* approach for detecting congestion rather than loss-based which takes a *reactive* approach after congestion already occurs. The advantage of delay over loss-based congestion control is that it detects the onset of congestion as switch buffers grow instead of waiting until they have filled for packet loss to occur. Delay-based congestion control infers congestion by comparing actual throughput to expected throughput. If actual throughput is significantly less than expected throughput, then the network is inferred to be congested.

Adapting to congestion. Congestion control adapts the sending rate based on the above congestion signals, typically accomplished by adjusting *cwnd* based on congestion severity. The most common approach for adjusting *cwnd* is the additive increase/multiplicative decrease (AIMD) scheme where *cwnd* linearly increases on each new ACK to probe for available bandwidth until a congestion event occurs, where *cwnd* exponentially decreases to “back off” from the network. When an RTO occurs, *cwnd* resets to 1 segment as this usually implies major changes in network conditions. Upon three duplicate acknowledgments, *cwnd* halves as this implies small amounts of packet loss. Fairness is achieved because flows back off at different times, allowing others to occupy the available bandwidth. The limitation of AIMD is while it can attain its fair share of bottleneck bandwidth, it backs off shortly after its discovery. The AIMD scheme is a high-level approach which varies depending on implementation.

E.3 A State Machine for Congestion Control

Acknowledgment packets. Every byte of data is associated with a unique sequence number [21] that increments by 1 with each additional data byte. A packet containing data is accompanied by a sequence number, which represents the sequence of the first byte of the data. Sequence numbers allow data segments to be easily reassembled and acknowledged by receivers. Acknowledgment packets are sent from the receiver

to explicitly inform the sender about the highest correctly received data byte thus far. They also implicitly inform the sender about current network conditions. An acknowledgment packet acknowledging sequence *X* implies all bytes up to but not including *X* have been correctly received. This allows the sender to either re-transmit lost segments or send new data. Receivers typically send one acknowledgment packet for every two received data packets.

Slow Start. This is the first state a connection enters and aims to quickly discover the available bandwidth of the network before congestion avoidance is entered. Since network capacities today span several orders of magnitude, this state performs an exponential search for the available bandwidth. When slow start is entered, *cwnd* starts at 1 MSS and doubles on every round-trip until either congestion is detected or *cwnd* reaches the target rate, the slow start threshold, or *ssthresh*. The slow start threshold defines the upper limit for *cwnd* growth while in slow start, which is initially set to the receive window, or *rwnd*.

Congestion Avoidance. The goal of this state is to avoid congestion by sending data at the estimated available bandwidth while slowly increasing *cwnd* to probe for available bandwidth. On every round-trip, *cwnd* increases by 1 MSS until congestion is detected: a RTO timeout or the recipient of three duplicate acknowledgments.

Fast Recovery. This state is entered from any other state when three duplicate acknowledgments are received. This event indicates less severe congestion because while it may indicate lost packets, it also indicates the network is at least able to transmit acknowledgments from the receiver. This state aims to quickly recover from the lost packets by halving *cwnd* and re-transmitting the last unacknowledged data segment. Fast recovery returns to congestion avoidance once all unacknowledged data before fast recovery was entered has been acknowledged.

Exponential Backoff. This state is entered when a re-transmission timeout (RTO) expires which infers major changes in network conditions. Re-transmission timers begin when data segments are first sent and expire when a certain amount of time has elapsed without the segment being acknowledged (usually several RTTs). Each time an RTO timeout expires, the segment is re-transmitted and the timer restarts with a doubled timeout time. This results in the sender exponentially backing-off from the network by allowing more time to elapse before it re-transmits the lost segment in response to the perceived network congestion. This repeats until an acknowledgment is received after which *ssthresh* becomes half of *cwnd*, *cwnd* restarts from 1 MSS and exponential-backoff transitions into slow start.

Cyber Threat Intelligence Modeling Based on Heterogeneous Graph Convolutional Network

Jun Zhao^{1,2}, Qiben Yan^{3,*}, Xudong Liu^{1,2,*}, Bo Li^{1,2,*}, Guangsheng Zuo^{1,2}

¹ School of Computer Science and Engineering, Beihang University, Beijing, China

² Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, China

³ Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA

Abstract

Cyber Threat Intelligence (CTI), as a collection of threat information, has been widely used in industry to defend against prevalent cyber attacks. CTI is commonly represented as Indicator of Compromise (IOC) for formalizing threat actors. However, current CTI studies pose three major limitations: first, the accuracy of IOC extraction is low; second, isolated IOC hardly depicts the comprehensive landscape of threat events; third, the interdependent relationships among heterogeneous IOCs, which can be leveraged to mine deep security insights, are unexplored. In this paper, we propose a novel CTI framework, HINTI, to model the interdependent relationships among heterogeneous IOCs to quantify their relevance. Specifically, we first propose multi-granular attention based IOC recognition method to boost the accuracy of IOC extraction. We then model the interdependent relationships among IOCs using a newly constructed heterogeneous information network (HIN). To explore intricate security knowledge, we propose a threat intelligence computing framework based on graph convolutional networks for effective knowledge discovery. Experimental results demonstrate that our proposed IOC extraction approach outperforms existing state-of-the-art methods, and HINTI can model and quantify the underlying relationships among heterogeneous IOCs, shedding new light on the evolving threat landscape.

1 Introduction

Nowadays, we are witnessing a rapid growth of sophisticated cyber attacks (e.g., zero-day attack, advanced persistent threat) [34]. Such attacks can effortlessly bypass traditional defenses such as firewalls and intrusion detection systems (IDS), breach critical infrastructures, and cause devastating catastrophes [7, 20, 39]. To combat these emerging threats, security experts proposed Cyber Threat Intelligence (CTI) that consists of a collection of Indicators of Compromise (IOCs). Different from the well-known secu-

rity databases (e.g., CVE¹, ExploitDB²), CTI can facilitate organizations to proactively release more comprehensive and valuable threat warnings (e.g., malicious IPs, malicious DNS, malware and attack patterns, etc.) when a system encounters suspicious outsider or insider threats [23].

In recent years, CTI has been increasingly adopted by security researchers and industries to share and capitalize on their discoveries, as well as by security firms to analyze the threat landscape using the deluge of data [5, 30]. The original CTI extraction and analysis require extensive manual inspection of the attack event descriptions, which becomes rather time-consuming given the enormous volume of threat description data. Recent studies have proposed automated methods to extract CTI in the form of Indicator of Compromise (IOC) from unstructured security-related texts [4, 22]. Most of existing IOC extraction methods, such as *CleanMX*³, *PhishTank*⁴, *IOC Finder*⁵, and *Gartner peer insight*⁶, follow the OpenIOC [10] standard and extract particular types of IOCs (e.g., malicious IP, malware, file Hash, etc) by leveraging a set of regular expressions. However, such extraction approaches face three major limitations. First, the accuracy of IOC extraction is low, which inevitably leads to the omission of critical threat objects [22]. Second, isolated IOC hardly depicts the comprehensive landscape of threat events, making it virtually impossible for CTI subscribers to gain a complete picture into the incoming threat. Third, there is a lack of an effective computing framework to efficiently measure the interactive relationships among heterogeneous IOCs.

To combat these limitations, HINTI, a cyber threat intelligence framework based on heterogeneous information network (HIN), is proposed to model and analyze CTIs. Specifically, HINTI proposes a multi-granular attention based IOC recognition approach to boost the accuracy of IOC extraction.

¹<http://cve.mitre.org/>

²<https://www.exploit-db.com/>

³<http://list.clean-mx.com>

⁴<https://www.phishtank.com>

⁵<https://www.fireeye.com/services/freeware/ioc-finder.html>

⁶<https://www.gartner.com/reviews/market/security-threat-intelligence-services>

Then, HINTI leverages HIN to model the interdependent relationships among heterogeneous IOCs, which can depict a more comprehensive picture of threat events. Moreover, we propose a novel CTI computing framework to quantify the interdependent relationships among IOCs, which helps uncover novel security insights. In short, the main contributions of this paper are summarized as follows:

- **Multi-granular Attention based IOC Recognition.** We propose multi-granular attention based IOC recognition approach to automatically extract cyber threat objects from multi-source threat texts, which can learn the significance of features with different scales. Our model outperforms the state-of-the-art methods in terms of IOC recognition accuracy and recall. In total, we extract over 397,730 IOCs from the unstructured threat descriptions.
- **Heterogeneous Threat Intelligence Modeling.** We model different types of IOCs using heterogeneous information network (HIN), which introduces various meta-paths to capture the interdependent relationships among heterogeneous IOCs while depicting a more comprehensive landscape of cyber threat events.
- **Threat Intelligence Computing Framework.** *We are the first* to present the concept of *cyber threat intelligence computing*, and design a general computing framework, as shown in Figure 5. The framework first utilizes a weight-learning based node similarity measure to quantify the interdependent relationships between heterogeneous IOCs, and then leverages attention mechanism based heterogeneous graph convolutional networks to embed the IOCs and their interactive relations.
- **Threat Intelligence Prototype System.** To evaluate the effectiveness of HINTI, we implement a CTI prototype system. Our system has identified 1,262,258 relationships among 6 types of IOCs including attackers, vulnerabilities, malicious files, attack types, devices and platforms, based on which we further assess the real-world applicability of HINTI using three real-world applications: IOC significance ranking, attack preference modeling, and vulnerability similarity analysis.

2 Background

2.1 Cyber Threat Intelligence

Cyber Threat Intelligence (CTI) extracted from security-related data is structured information used to proactively resist cyber attacks. CTI consists of reasoning, context, mechanism, indicators, implications, and actionable advice about an existing or evolving cyber attack that can be used to create preventive measures in advance [30]. CTI allows subscribers to expand their visibility into the fast-growing threat landscape, and enable early identification and prevention of a

cyber threat. Take WannaCry virus as an example, if security guards can timely capture the threat intelligence that indicates “Wannacry permeates port 445 to attack systems”, the malicious intrusion can be easily blocked by locking down port 445, which is the most direct and effective way of combating WannaCry virus [7].

Meanwhile, social media (e.g., Blog, Twitter) has increasingly become an effective medium for exchanging and spreading cyber security information, on which security experts and organizations often post their discoveries to reach a wider audience promptly [32]. These posts usually include a magnitude of valuable security-related information [25, 26], such as the warnings regarding latest vulnerabilities, hacking tools, data breaches, and existing or upcoming software patches, providing one of the main raw materials for extracting CTIs.

Early CTI extraction requires extensive manual inspection of the threat descriptions, which becomes rather time-consuming given the enormous volume of such descriptions. To facilitate the automatic generation and sharing of CTI, a large volume of methods and frameworks are established, such as *IODEF* [13], *STIX* [4], *TAXII* [36], *OpenIOC* [10], and *CyBox* [28], CleanMX, PhishTank, IOC Finder and [2, 22, 31, 46]. The majority of existing methods and frameworks leverage regular expressions to extract IOCs, which may suffer from a low accuracy due to their inability in pre-defining a comprehensive set of the IOC rules.

2.2 Motivation

The main goal of this research is to address the limitations of the existing CTI analytics frameworks by modeling the interdependent relationships among heterogeneous IOCs. As a motivating example, given a security-related post: “*Last week, Lotus exploited CVE-2017-0143 vulnerability to affect a larger number of Vista SP2 and Win7 SP devices in Iran. CVE-2017-0143 is a remote code execution vulnerability including a malicious file SMB.bat*”. Most of the existing CTI frameworks can extract specific IOCs but neglect the relationships among them, as shown in Figure 1. It is obvious that such IOCs could not draw a comprehensive picture of the threat landscape, let alone quantifying their interactive relationships for in-depth security investigation.

Different from the existing CTI frameworks, HINTI aims to implement a computational CTI framework, which can not only extract IOCs efficiently but also model and quantify the relationships between them. Here, we use the motivating example to illustrate how HINTI works step-by-step in practice as follows.

(i) First, the security-related post is annotated by the *B-I-O* sequence tagging method [43] as shown in Figure 2, where *B-X* indicates that the element of type *X* is located at the beginning of the fragment, *I-X* means that the element belonging to type *X* is located in the middle of the fragment, and *O* represents a non-essential element of other types. In this

```

<? Xml version=1.0 encoding=utf-8>
<indicator id=1a0ee12, op=OR>
  <Description>
    <Actor>Lotus</Actor>
    <Vul>CVE-2017-0143</Vul>
    <Dev>Vista SP2</Dev>
    <Dev>Win7 SP1</Dev>
    <Type>Remote code execution</Type>
    <File>SMB.bat</File>
  </Description>

```

Figure 1: An example of extracted IOCs without any relations among them.

research, we annotated 30,000 such training samples from 5,000 threat description texts, which are the raw materials used to build our IOC extraction model.

```

Last(O) week(O), Lotus(B-Attacker) exploited(O) CVE-2017-0143(B-Vul) vulnerability(O) to(O) affect(O) a(O) large(O) number(O) of(O) Vista SP2(B-Device) and(O) Win7 SP1(B-Device) devices(O) in(O) Iran(O). CVE-2017-0143(B-Vul) is(O) a(O) remote(B-Type) code(I-Type) execution(I-Type) vulnerability(O) involving(O) a(O) malicious(O) file(O) SMB.bat(B-File).

```

Figure 2: An annotation example with the *B-I-O* tagging method.

(ii) The labeled training samples are then fed into the proposed neural network architecture as shown in Figure 6 to train our proposed IOC extraction model. As a result, HINTI has the ability to accurately identify and extract IOCs (e.g., *Lotus*, *SMB.bat*) using the proposed multi-granular attention based IOC extraction method (see Section 4.1 for details).

(iii) HINTI then utilizes the syntactic dependency parser [6] (e.g., subject-predicate-object, attributive clause, etc.) to extract associated relationships between IOCs, each of which is represented as a triple $(IOC_i, relation, IOC_j)$. In this motivating example, HINTI extracts the relationship triples involving $(Lotus, exploit, CVE - 2017 - 0143)$, $(CVE - 2017 - 0143, affect, VistaSP2)$, etc. Note that the extracted relational triples can be incrementally pooled into an HIN to model the interactions among IOCs for depicting a more comprehensive threat landscape. Figure 3 shows a miniature graphic representation describing interactive relations among IOCs extracted from the example. Compared with Figure 1, it is obvious that HINTI can depict a more intuitive and comprehensive threat landscape than the previous approaches. In this paper, we mainly consider 9 relationships (R1~R9) among 6 different types of IOCs (see Section 4.2 for details).

(iv) Finally, HINTI integrates a CTI computing framework

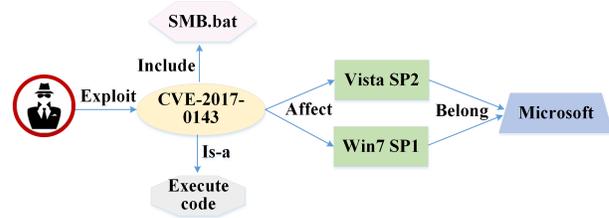


Figure 3: A miniature of a constructed CTI includes attacker, vulnerability, malicious file, attack type, device, and platform, which describes a particular threat: an attacker utilizes *CVE-2017-0143* vulnerability to invade *Vista SP2* and *Win7 SP1* devices. *CVE-2017-0143* is a *remote code execution* vulnerability that involves a malicious file “*SMB.bat*”.

based on heterogeneous graph convolutional networks (see Section 4.3) to effectively quantify the relationships among IOCs for knowledge discovery. Particularly, our proposed CTI computing framework characterizes IOCs and their relationships in a low-dimensional embedding space, based on which CTI subscribers can use any classification (e.g., *SVM*, *Naive Bayes*) or clustering algorithms (*K-Means*, *DBSCAN*) to gain new threat insights, such as predicting which attackers are likely to intrude their systems, and identifying which vulnerabilities belong to the same category without the expert knowledge. In this work, we mainly explore three real-world applications to verify the effectiveness and efficiency of the CTI computing framework: IOC significance ranking (see Section 6.1), attack preference modeling (see Section 6.2), and vulnerability similarity analysis (see Section 6.3).

2.3 Preliminaries

In this paper, we use heterogeneous information network (HIN) to model the relationships among IOCs. Here, we first introduce the preliminary knowledge about HIN.

Definition 1 Heterogeneous Information Network of Threat Intelligence (HINTI) is defined as a directed graph $G = (V, E, T)$ with an object type mapping function $\varphi : V \rightarrow M$ and a link type mapping function $\Psi : E \rightarrow R$. Each object $v \in V$ belongs to one particular object type in the object type set M : $\varphi(v) \in M$, and each link $e \in E$ belongs to a particular relation type in the relation type set R : $\Psi(e) \in R$. T denotes the types of nodes and relationships.

In this paper, we focus on 6 common types of IOCs: *attacker* (A), *vulnerability* (V), *device* (D), *platform* (P), *malicious file* (F), and *attack type* (T), and the links connecting different objects represent different semantic relationships. To better understand the object types and relationship types in HINTI, it is imperative to provide the meta-level (i.e., schema-level) description of the network. Consequently, we introduce

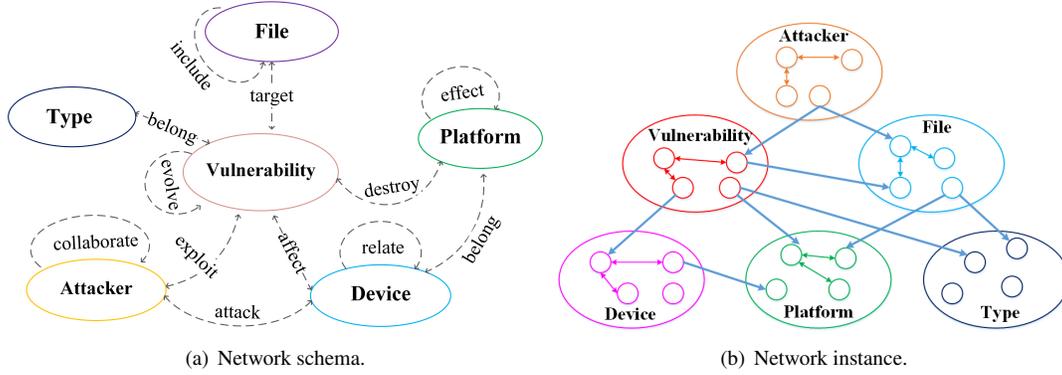


Figure 4: Network schema and instance of HIN containing 6 types of IOCs. (a): The network schema of HIN, which depicts the relationship template among different types of IOCs, such as $Device \xrightarrow{belong} Platform$. (b): An instance of network schema, which describes the concrete relationships between IOCs by following a network schema, e.g., $Office\ 2012 \xrightarrow{belong} Windows$.

the network schema [37] for describing the meta-level relationships.

Definition 2 Network Schema. *The network schema of HINTI, denoted as $H_S = (A, R)$, is a meta template for $G = (V, E, T)$ with the object type mapping $\phi : V \rightarrow M$ and the link type mapping $\Phi : E \rightarrow R$. It is a directed graph of object types M with edges representing relations from R .*

The schema of HINTI specifies type constraints on the sets of IOCs and their relationships. Figure 4 (a) shows the network schema of HINTI, which defines the relationship templates among IOCs to effectively guide the semantic exploration in HINTI. For example, for a relationship that describes: “attackers invade devices”, the semantic schema can be written as: $attacker \xrightarrow{invade} device$. Figure 4 (b) presents a concrete instance of the network schema.

Definition 3 Meta-path. *A meta-path [37] P is a path sequence defined on a network schema $S = (N, R)$, and is represented in the form of $N_1 \xrightarrow{R_1} N_2 \xrightarrow{R_2} \dots \xrightarrow{R_i} N_{i+1}$, which defines a composite relation $R = R_1 \diamond R_2 \diamond \dots \diamond R_{i+1}$, where \diamond denotes the composition operator on relations. A meta-path P is a symmetric path when the relation R defined by the path is symmetric (i.e., P is equal to P^{-1}).*

Table 1 displays the meta-paths considered in HINTI. For example, the relationship “the attackers (A) exploit the same vulnerability (V)” can be described by a length-2 meta-path $attacker \xrightarrow{exploit} vulnerability \xrightarrow{exploit^-} attacker$, denoted as AVA^T (P_4), which means that the two attackers exploit the same vulnerability. Similarly, $AVDPD^T V^T A^T$ (P_{17}) portrays a close relationship between IOCs that “two attackers who leverage the same vulnerability invade the same type of device and ultimately destroy the same type of platform”.

Table 1: Meta-paths used in HINTI.

ID	Meta-path
P_1	Attacker-Attacker
P_2	Device-Device
P_3	Vulnerability-Vulnerability
P_4	Attacker-Vulnerability-Attacker
P_5	Attacker-Device-Attacker
P_6	Device-File-Device
P_7	Device-Platform-Device
P_8	Vulnerability-File-Vulnerability
P_9	Vulnerability-Type-Vulnerability
P_{10}	Vulnerability-Device-Vulnerability
P_{11}	Vulnerability-Platform-Vulnerability
P_{12}	Attacker-Device-Platform-Device-Attacker
P_{13}	Attacker-Vul-Device-Vul-Attacker
P_{14}	Attacker-Vul-Platform-Vul-Attacker
P_{15}	Attacker-Vul-Type-Vul-Attacker
P_{16}	Vul-Device-Platform-Device-Vul
P_{17}	Attacker-Vul-Device-Platform-Device-Vul-Attacker

3 Architecture Overview of HINTI

HINTI, as a cyber threat intelligence extraction and computing framework, is capable of effectively extracting IOCs from threat-related descriptions and formalizing the relationships among heterogeneous IOCs to demystify new threat insights. As shown in Figure 5, HINTI consists of four major components, including:

- **Data Collection and IOC Recognition.** We first de-

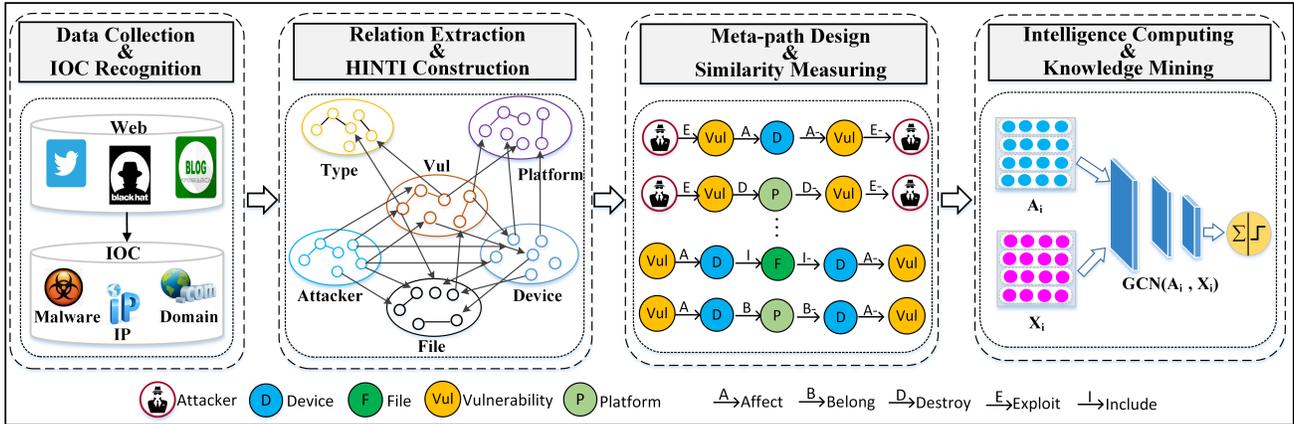


Figure 5: The overall architecture of HINTI. HINTI consists of four major components: (a) collecting security-related data and extracting threat objects (i.e., IOCs); (b) modeling interdependent relationships among IOCs into a heterogeneous information network; (c) embedding nodes into a low-dimensional vector space using weight-learning based similarity measure; and (d) computing threat intelligence based on graph convolutional networks and knowledge mining.

velop a data collection system to automatically capture security-related data from blogs, hacker forum posts, security news, and security bulletins. The system utilizes a breadth-first search to collect the HTML source code, and then leverages Xpath (XML Path language) to extract threat-related descriptions. After that, we utilize a multi-granular attention based IOC recognition method to extract IOC from the collected threat-related descriptions (see Section 4.1 for details).

- **Relation Extraction and IOC modeling.** HINTI addresses the challenge of CTI modeling by leveraging heterogeneous information networks, which can naturally depict the interdependent relationships between heterogeneous IOCs. As an example, Figure 4 shows a model that capture the interactive relationships among attacker, vulnerability, malicious file, attack type, platform, and device (see Section 4.2 for details).
- **Meta-path Design and Similarity Measure.** Meta-path is an effective tool to express the semantic relations among IOCs in constructed HIN. For instance, $attacker \xrightarrow{exploit} vulnerability \xrightarrow{exploit^-} attacker$, indicates that two attackers are related by exploiting the same vulnerability. We design 17 types of meta-paths (See Table 1) to describe the interdependent relationships between IOCs. With these meta-paths, we present a weight-learning based node similarity computing approach to quantify and embed the relationships as the premise for threat intelligence computing.
- **Threat Computing and Knowledge Mining.** In this component, an effective threat intelligence computing framework is proposed, which can quantify and measure

the relevance among IOCs by leveraging graph convolutional network (GCN). Our proposed threat intelligence computing framework could reveal richer security knowledge within a more comprehensive threat landscape.

4 Methodology

4.1 Multi-granular Attention Based IOC Extraction

Extracting IOCs from multi-source threat texts is one of the major tasks of threat intelligence analytics, and the quality of the extracted IOCs significantly influences the analysis results of cyber threats. Recently, Bidirectional Long Short-Term Memory+Conditional Random Fields (BiLSTM+CRF) model [15] has demonstrated excellent performance in text chunking and Named-entity Recognition (NER). However, directly applying this model to IOC extraction is unlikely to succeed, since threat texts usually contain a large number of threat objects with different grams and irregular structures. Consequently, we need an efficient method to learn the discriminative characteristics of IOCs with different sizes. In this paper, we propose a multi-granular attention based IOC extraction method, which can extract threat objects with different granularity. Particularly, Figure 6 presents the proposed IOC extraction framework, which leverages the multi-granular attention mechanism to characterize IOCs. Different from the traditional BiLSTM+CRF model, we introduce new word-embedding features with different granularities to capture the characteristics of IOCs with different sizes. Furthermore, we utilize a self-attention mechanism to learn the importance of the features to improve the accuracy of IOC extraction.

Our proposed method takes a threat description sentence $X = (x_1, x_2, \dots, x_i)$ as input, where x_i represents i -th word

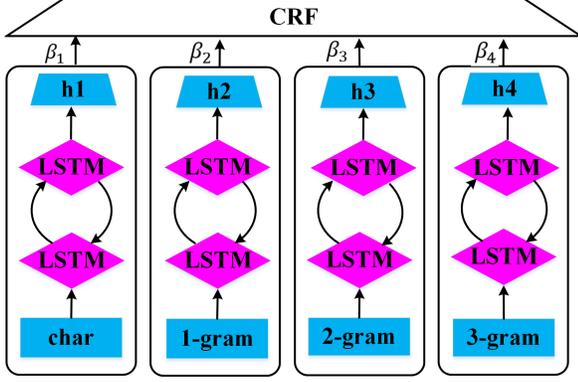


Figure 6: The framework of multi-granular IOC extraction.

in X . We first chunk the sentence into n-gram components including char-level, 1-gram, 2-gram, and 3-gram, which are the inputs of our trained model, written as follows:

$$e_{x_i}^j = V_{embedding}^j(x_i), \quad (1)$$

where $V_{embedding}^j$ transforms the chunk with granularity j into a vector space and x_i is the i -th word in a sentence X . Thus, the threat description sentence X_i can be vectorized as follows:

$$\begin{aligned} \vec{h}_i^j &= LSTM_{forward}([e_{x_0}^j, e_{x_1}^j, \dots, e_{x_i}^j]) \\ \overleftarrow{h}_i^j &= LSTM_{backward}([e_{x_0}^j, e_{x_1}^j, \dots, e_{x_i}^j]) \end{aligned} \quad (2)$$

where \vec{h}_i^j and \overleftarrow{h}_i^j are the embedded features learned by forward LSTM and backward LSTM, respectively. Let O be the output of Bi-LSTM, which is a weighted sum of embedded features with weights corresponding to the importance of different features:

$$O = H \cdot W^T \quad (3)$$

where $H = \sum_j \vec{\beta}_j \sigma(h_1^j, h_2^j, \dots, h_i^j)$, $h_i^j = (\vec{h}_i^j + \overleftarrow{h}_i^j)$, $\vec{\beta}_j$ is the weight vector to represent the importance of h_i^j , in which j and i are the segmentation granularity of sentences and the corresponding index of the chunk. W is the parameter matrix.

Given a security-related sentence $X = (x_1, x_2, \dots, x_i)$, its corresponding threat object sequence $Y = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_i)$, and its output of Bi-LSTM O , we can compute the overall label score of X and Y as follows:

$$S(X, Y) = \sum_{i=0}^n (A_{\hat{y}_i, \hat{y}_{i+1}} + O_{i, \hat{y}_i}) \quad (4)$$

where $A_{\hat{y}_i, \hat{y}_{i+1}}$ is the state transition matrix in CRF model, and O_{i, \hat{y}_i} , as the output of Bi-LSTM hidden layer (calculated by Eq. (3)), represents the label score of i -th word corresponding

to the type \hat{y}_i . Next, we utilize *softmax* function to normalize the overall label score:

$$p(Y|X) = \frac{e^{S(X, Y)}}{\sum_{\tilde{y} \in Y_X} e^{S(X, \tilde{y})}} \quad (5)$$

We design an objective function to maximize the probability $p(Y|X)$ to achieve the highest label score for different IOCs, which can be written as follows:

$$\begin{aligned} \text{argmax} \log(p(Y|X)) &= \text{argmax} (S(X, Y) - \\ &\log(\sum_{\tilde{y} \in Y_X} e^{S(X, \tilde{y})})) \end{aligned} \quad (6)$$

By solving the objective function above, we assign correct labels to the n-gram components, according to which we can identify the IOCs with different lengths. Our multi-granular attention based IOC extraction method is capable of identifying different types of IOCs, and its evaluation is presented in Section 5.

4.2 Cyber Threat Intelligence Modeling

CTI modeling is an important step to explore the intricate relationship between heterogeneous IOCs. In our work, HIN is introduced to group different types of IOCs into a graph to explore their interactive relationships. In this section, we portray the main principle of threat intelligence modeling.

To model the intricate interdependent relationships among IOCs, we define the following 9 relationships among 6 types of IOCs as follows.

- **R1:** To depict the relation of an attacker and the exploited vulnerability, we construct the *attacker-exploit-vulnerability* matrix A . For each element $A_{i,j} \in \{0, 1\}$, $A_{i,j}=1$ indicates attacker i exploits vulnerability j .
- **R2:** To depict the relation of an attacker and a device, we build the *attacker-invade-device* matrix D . For each element $D_{i,j} \in \{0, 1\}$, $D_{i,j}=1$ indicates attacker i invades device j .
- **R3:** Two attacker can cooperate to attack a target. To study the relationship of *attacker-attacker*, we construct the *attacker-cooperate-attacker* matrix C . For each element $C_{i,j} \in \{0, 1\}$, $C_{i,j}=1$ indicates there exists a cooperative relationship between attacker i and j .
- **R4:** To describe the relation of a vulnerability and the affected device, we build the *vulnerability-affect-device* matrix M . For each element $M_{i,j} \in \{0, 1\}$, $M_{i,j}=1$ indicates vulnerability i affects device j .
- **R5:** A vulnerability is often labeled as a specific attack type by Common Vulnerabilities and Exposures (CVE)

system⁷. To explore the relation of *vulnerability-attack type*, we build the *vulnerability-belong-attack type* matrix G , where each element $G_{i,j} \in \{0, 1\}$ denotes if vulnerability i belongs to an attack type j .

- **R6:** A vulnerability often involves one or more malicious files. To describe the relation of *vulnerability-file*, we build the *vulnerability-include-file* matrix F . For each element $F_{i,j} \in \{0, 1\}$, $F_{i,j}=1$ denotes that vulnerability i includes malicious file j .
- **R7:** A malicious file often targets a specific device. We establish the *file-target-device* matrix T to explore the relation of *file-device*. For each element $T_{i,j} \in \{0, 1\}$, $T_{i,j}=1$ indicates malicious file i targets device j .
- **R8:** Oftentimes, a vulnerability evolves from another. To study the relationship of *vulnerability-vulnerability*, we build the *vulnerability-evolve-vulnerability* matrix E , where each element $E_{i,j} \in \{0, 1\}$ indicates if vulnerability i evolves from vulnerability j .
- **R9:** To depict the relation *device-platform* that a device belongs to a platform, we build the *device-belong-platform* matrix P where each element $P_{i,j} \in \{0, 1\}$ illustrates if device i belongs to platform j .

Based on the above 9 types of relationships, HINTI leverages the syntactic dependency parser [6] (e.g., subject-predicate-object, attributive clause, etc.) to automatically extract the 9 relationships among IOCs from threat descriptions, each of which is represented as a triple ($IOC_i, relation, IOC_j$). For instance, given a security-related description: "On May 12, 2017, WannaCry exploited the MS17-010 vulnerability to affect a larger number of Windows devices, which is a ransomware attack via encrypted disks". Using the syntactic dependency parser, we can extract the following triples: (WannaCry, exploit, MS17-010), (MS17-010, affect, Windows device), (WannaCry, is, ransomware). Such triples extracted from various data sources can be incrementally assembled into HINTI to model the relationships among IOCs, which could offer a more comprehensive threat landscape that describes the threat context. Particularly, we further define 17 types of meta-paths shown in Table 1 to probe into the interdependent relationships over attackers, vulnerabilities, malicious files, attack types, devices, and platforms. HINTI is able to convey a richer context of threat events by scrutinizing 17 types of meta-paths and reveal the in-depth threat insights behind the heterogeneous IOCs (see Section 6 for details).

4.3 Threat Intelligence Computing

In this section, we illustrate the concept of threat intelligence computing, and design a general threat intelligence computing

⁷<http://cve.mitre.org/>

framework based on heterogeneous graph convolutional networks, which quantifies and measures the relevance between IOCs by analyzing meta-path based semantic similarity. Here, we first provide a formal definition of threat intelligence computing based on heterogeneous graph convolutional networks.

Definition 4 Threat Intelligence Computing Based on Heterogeneous Graph Convolutional Networks. Given the threat intelligence graph $G = (V, E)$, the meta-path set $M = \{P_1, P_2, \dots, P_l\}$. The threat intelligence computing: i) computes the similarity between IOCs based on meta-path P_i ; to generate corresponding adjacency matrix A_i ; ii) constructs the feature matrix of nodes X_i by embedding attribute information of IOCs into a latent vector space; iii) conducts graph convolution $GCN(A_i, X_i)$ to quantify the interdependent relationships between IOCs by following meta-path P_i , and embeds them into a low-dimensional space.

The threat intelligence computing aims to model the semantic relationships between IOCs and measure their similarity based on meta-paths, which can be used for advanced security knowledge discovery, such as threat object classification, threat type matching, threat evolution analysis, etc. Intuitively, the objects connected by the most significant meta-paths tend to bear more similarity [37]. In this paper, we propose a weight-learning based threat intelligence similarity measure, which uses self-attention to improve the performance of similarity measurement between any two IOCs. This method can be formalized as below:

Definition 5 Weight-learning based Node Similarity Measure. Given a set of symmetric meta-path set $P = [P_m]_{m=1}^M$, the similarity $S(h_i, h_j)$ between any two IOCs h_i and h_j is defined as:

$$S(h_i, h_j) = \sum_m \vec{w}_m \frac{2 \cdot |\{h_{i \rightarrow j} \in P_m\}|}{|\{h_{i \rightarrow i} \in P_m\}| + |\{h_{j \rightarrow j} \in P_m\}|} \quad (7)$$

where $h_{i \rightarrow j} \in h_m$ is a path instance between IOC h_i and h_j following meta-path P_m , $h_{i \rightarrow i} \in P_m$ is that between IOC instance h_i and h_i , and $h_{j \rightarrow j} \in P_m$ is that between IOC instance h_j and h_j , where $|\{h_{i \rightarrow j} \in P_m\}| = C_{P_m}(i, j)$, $|\{h_{i \rightarrow i} \in P_m\}| = C_{P_m}(i, i)$, $|\{h_{j \rightarrow j} \in P_m\}| = C_{P_m}(j, j)$, and C_{P_m} is the commuting matrix based on meta-path P_m defined below. $\vec{w} = [w_1, \dots, w_m, \dots, w_M]$ denote the meta-path weights, where w_m is the weight of meta-paths P_m , and M is the number of meta-paths.

$S(h_i, h_j)$ is defined in two parts: (1) the semantic overlap in the numerator, which describes the number of meta-path between IOC instance h_i and h_j ; (2) and the semantic broadness in the denominator, which depicts the number of total meta-paths between themselves. The larger number of meta-path between IOC instance h_i and h_j , the more similar the two IOCs are, which is normalized by the semantic broadness

of denominator. Moreover, different from existing similarity measures [37], we propose an attention mechanism based similarity measure method by introducing the weight vector $\vec{w} = [w_1, \dots, w_m, \dots, w_{M'}]$, which is a trainable coefficient vector to learn the importance of different meta-paths for characterizing IOCs.

Obviously, it is computationally expensive to measure the similarity among IOCs in the constructed heterogeneous graph as it usually requires to randomly walk a larger number of nodes in the graph. Fortunately, in our work, it is unnecessary to walk through the entire graph as we prescribe a limit by introducing predefined meta-paths, and we only focus on the symmetrical meta-paths presented in Table 1. To calculate the similarity between IOCs under different meta-path instances, we need to compute the corresponding commuting matrices [37] following the meta-paths.

Given a meta-path set $P = \sum_m^M \{A_1, A_2, \dots, A_{l+1}\}$, the meta-path based commuting matrix can be defined as $C_P = U_{A_1 A_2} \circ U_{A_2 A_3} \dots \circ U_{A_l A_{l+1}}$, where $C_P(i, j)$ represents the probability of object $i \in A_1$ reaching object $j \in A_{l+1}$ under the path P , and \circ is a connection operation. These symmetric meta-paths not only efficiently reduce the complexity of walking, but also ensures that the commuting matrix can be easily decomposed, which greatly reduces the computational costs. In addition, the symmetric meta-paths in the graph G allow us to leverage the pairwise random-walk [37] to further accelerate the computation.

With Eq. (7) and pairwise random-walk, we can obtain the similarity embedding between any two IOCs h_i and h_j under a meta-path set P . Based on the low-dimensional similarity embedding, we derive a weighted adjacent matrix between IOCs, denoted as $A_i \in R^{N \times N}$, where N is the number of a specific type of IOC in G . Meanwhile, to utilize the attributed information of nodes, we train a word2vec model [24] to embed the attribute information of nodes into a feature matrix $X_i \in R^{N \times d}$, where N is the number of IOCs in A_i , and d is the dimension of node feature. With the learned adjacency matrix A_i and its feature matrix X_i , we can leverage the classical GCN [18] to characterize the relationship between IOC h_i and h_j . Particularly, the layer-wise propagation rule of GCN can be defined as below:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}) \quad (8)$$

where $\tilde{A} = A + I_N$ is the adjacency matrix of IOCs with self-connections, I_N is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, and $W^{(l)}$ is a l -th layer trainable weight matrix. $\sigma(\cdot)$ denotes an activation function, such as *relu*. $H^{(l)} \in R^{N \times d}$ is the matrix of activation in the l -th layer. We perform graph convolution [18] on A_i and X_i to generate the embedding Z between IOCs belonging to type i :

$$Z = f(X_i, A_i) = \sigma(\hat{A}_i \cdot \text{relu}(\hat{A}_i X_i W_i^{(0)})) W_i^{(1)} \quad (9)$$

where $W_i^{(0)} \in R^{d \times H}$ is an input-to-hidden weight matrix for a

hidden layer with H feature maps, $W_i^{(1)} \in R^{H \times F}$ is a hidden-to-output weight matrix with F feature maps in the output layer, $X_i \in R^{N \times d}$, N is the number of a specific type of IOCs, d is the dimension of their corresponding features, and σ is another activation function, such as *sigmoid*. $\hat{A}_i = \tilde{D}^{-\frac{1}{2}} \tilde{A}_i \tilde{D}^{-\frac{1}{2}}$ can be calculated offline. Here, we leverage the cross-entropy loss to optimize the performance of our proposed threat intelligence framework, written as follows:

$$\text{Loss}(Y_{lf}, Z_{lf}) = - \sum_{i \in Y_{lf}} \sum_{f=1}^F Y_{lf} \cdot \ln Z_{lf} \quad (10)$$

where Y_l is the set of node indices that have labels, Y_{lf} is the real label, and Z_{lf} is a corresponding label that our model predicts. Based on Eq. (10), we conduct stochastic gradient descent to continuously optimize the neural network weights $W_i^{(0)}$, $W_i^{(1)}$, and \vec{w} to reduce the loss, and build a general threat intelligence computing framework. Using this framework, security organizations are able to mine richer security knowledge hidden in the interdependent relationships among IOCs.

5 Experimental Evaluation

5.1 Dataset and Settings

We develop a threat data collector to automatically collect cyber threat data from a set of sources, including 73 international security blogs (e.g., *fireeye*, *cloudflare*), hacker forum posts (e.g., *Blackhat*, *Hack5*), security bulletins (e.g., *Microsoft*, *Cisco*), CVE detail description, and ExploitDB. A complete list of data sources is presented in the Baidu cloud⁸. We set up a daemon to collect the newly generated security events every day. So far, more than 245,786 security-related data describing threat events have been collected. For training and evaluating our proposed IOC extraction method, 30,000 samples from 5,000 texts are annotated by utilizing the *B-I-O* sequence tagging method (see Section 2.2 for the example), and an annotation example is shown in Figure 2.

For 30,000 labeled samples, we randomly select 60% of samples as a training set, 20% of samples as a verification set, and the rest of the samples as our test set. Based on the data sets, we comprehensively evaluate the performance of HINTI for extracting IOCs and threat intelligence computing. We run all of the experiments on 16 cores Intel(R) Core(TM) i7-6700 CPU @3.40GHz with 64GB RAM and 4 × NVIDIA Tesla K80 GPU. The software programs are executed on the TensorFlow-GPU framework on Ubuntu 16.0.4.

5.2 Evaluation of IOC Extraction

A set of experiments are conducted to evaluate the sensitivity of different parameters in the multi-granular based IOC

⁸https://pan.baidu.com/s/1J631WMYT_awa8aY5xy3A

extraction model. We mainly consider 8 hyper-parameters that seriously impact the performance of the model as shown in Table 2. More specifically, `Embedding_dim` is one of the most important factors that determine the generalization capability of the model. Here, we fix other parameters while fine-tuning the embedding size in the range of (50, 100, 150, 200, 250, 300, 350, 400). Experimental results show that the accuracy of extracted IOC achieves the best when `Embedding_dim=300`. `Learning_rate` is another major factor for determining the stride of gradient descent in minimizing the loss function, which determines whether the model can find a global optimal solution. We fix other parameters to fine-tune the `Learning_rate` in the range of (0.001, 0.005, 0.01, 0.05, 0.1, 0.5), and the performance reaches the best when the `Learning_rate=0.001`. Similarly, we fine-tune the other hyper-parameters with 5,000 epochs, and the hyper-parameters allowing our model to perform optimally are recorded in Table 2.

Table 2: Hyperparameters setting in the multi-granular based IOC extraction method.

Parameter	value	Parameter	Value
Embedding_dim	300	Hidden_dim	128
Sequence_length	500	Epoch_num	5,000
Learning_rate	0.001	Batch_size	64
Dropout_rate	0.5	Optimizer	Adam

Table 3: Performance of IOC extraction w.r.t. IOC types.

IOC Type	Precision	Recall	Micro-F1
<i>IP</i>	99.56	99.52	99.54
<i>File</i>	94.36	96.88	95.60
<i>Type</i>	99.86	99.81	99.83
<i>Email</i>	99.32	99.87	99.49
<i>Device</i>	93.26	92.78	93.02
<i>Vender</i>	93.07	94.45	94.24
<i>Version</i>	96.98	97.99	97.48
<i>Domain</i>	96.58	95.89	96.23
<i>Software</i>	88.25	89.31	88.78
<i>Function</i>	95.03	95.59	95.31
<i>Platform</i>	94.31	92.57	93.43
<i>Malware</i>	89.76	91.23	90.49
<i>Vulnerability</i>	99.25	98.73	98.99
<i>Other</i>	98.29	98.42	98.35

In this paper, we extract 13 major types of IOCs, and the

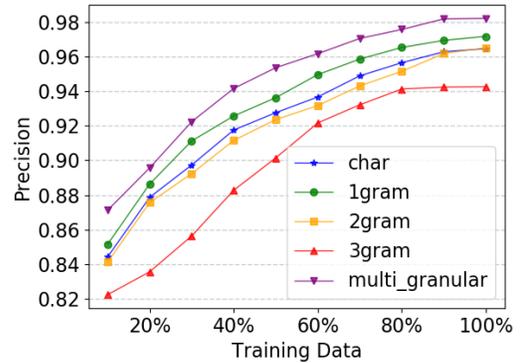


Figure 7: Performance of IOC extraction using embedding features with different granularity.

performance is presented in Table 3. Overall, our IOC extraction method demonstrates excellent performance in terms of precision, recall, and Micro-F1 (i.e., micro-averaged F1-score) for most types of IOCs, such as *function*, *malicious IP*, and *device*. However, we observe a performance degradation when recognizing software and malware. This can be attributed to the fact that most software and malware is named by random strings such as md5 hash. Moreover, we find that the number of training samples impacts the performance of the model. Specifically, the performance becomes unsatisfactory (e.g., *Software*, *Malware*) when the number of a certain type of training samples is insufficient (i.e., less than 5,000).

In order to verify the effectiveness of multi-granular embedding features, we assess the performance of IOC extraction with features of different granularity including *char-level*, *1-gram*, *2-gram*, *3-gram* and *multi-granular* features. The experimental results are demonstrated in Figure 7, from which we can observe that the proposed multi-granular embedding feature outperforms others since it leverages the attention mechanism to simultaneously learn multi-granular features to characterize different patterns of IOCs.

Next, to verify the effectiveness of the proposed IOC extraction method, we compare it with the state-of-the-art entity recognition approaches, including general NER tools *NLTK NER*⁹, and *Stanford NER*¹⁰, professional IOC extraction method *Stucco* [16] and *iACE* [22], and popular entity recognition approaches *CRF* [21], *BiLSTM* and *BiLSTM+CRF* [15]. The experimental results of different methods on real-world data are demonstrated in Table 4.

The results indicate that our proposed IOC extraction outperforms the state-of-the-art entity recognition methods and tools in terms of precision, recall, and Micro-F1, and its improvement can be attributed to the following factors. First, compared with *Stanford NER* and *NLTK NER*, the NLP tools

⁹<http://www.nltk.org/book/ch07.html>

¹⁰<https://stanfordnlp.github.io/CoreNLP/ner.html>

Table 4: Performance of threat entity recognition using different methods.

Method	Accuracy	Precision	Micro-F1
<i>NLTK NER</i>	69.45	68.51	67.49
<i>Stanford NER</i>	68.35	66.74	68.58
<i>iACE</i>	92.14	91.26	92.25
<i>Stucco</i>	91.16	92.21	91.47
<i>CRF</i>	92.64	91.80	92.65
<i>BiLSTM</i>	94.78	95.21	94.35
<i>BiLSTM+CRF</i>	96.38	96.42	96.27
<i>Multi-granular</i>	98.59	98.72	98.69

trained with general news corpora, our method uses a security-related training corpus collected and labeled by ourselves as a data source for training our model. Second, different from the rule-based extraction approaches (e.g., *iACE* and *Stucco*), our proposed deep learning based method provides an end-to-end system with more advanced features to represent various IOCs. Third, comparing to RNN-based methods (e.g., *BiLSTM* and *BiLSTM+CRF*), our proposed method brings in multi-granular embedding sizes (*char-level*, *1-gram*, *2-gram*, and *3-gram*) to simultaneously learn the characteristics of various sizes and types of IOCs, which can identify more complex and irregular IOCs. Last but not the least, our method implements an attention mechanism to learn the weights of features with various scales to effectively characterize different types of IOCs, further enhancing the IOC recognition accuracy.

6 Application of Threat Intelligence Computing

Our proposed threat intelligence computing framework based on heterogeneous graph convolutional networks can be used to mine novel security knowledge behind heterogeneous IOCs. In this section, we evaluate its effectiveness and applicability using three real-world applications: profiling and ranking for CTIs, attack preference modeling, and vulnerability similarity analysis.

6.1 Threat Profiling and Significance Ranking of IOCs

Due to the disparity in the significance of threats, it is important to derive the threat profile and rank the significance of IOCs for demystifying the landscape of threats. However, most of the existing CTIs are incapable of modeling the associated relationships between heterogeneous IOCs.

Different from isolated CTIs, HINTI leverages HIN to

model the interdependent relationships among IOCs with two characteristics: first, the isolated IOCs can be integrated into a graph-based HIN to clearly display the associated relationships among IOCs, which is capable of directly depicting the basic threat profile. For example, Figure 3 depicts a threat profiling sample: an attacker utilizes *CVE-2017-0143* vulnerability to invade *Vista SP2* and *Win7 SP1* devices belonging to the Microsoft platform, and *CVE-2017-0143* is a *remote code execution* vulnerability that uses a “*SMB.bat*” malicious file. Second, the significance of IOCs in HINTI can be naturally ranked based on the proposed threat intelligence computing framework.

Table 5 shows the top 5 authoritative ranking score [35] of vulnerability, attacker, attack type, and platform, from which security experts can gain a clear insight into the impact of each IOC. Degree centrality [33], which describes the number of links incident upon a node, is widely used in evaluating the importance of a node in a graph. It can be used to quantify the immediate risk of a node that connects with other nodes for delivering network flows, such as virus spreading. Here, degree centrality can be utilized in verifying the effectiveness of the proposed threat intelligence computing framework in ranking the importance of IOCs. It is worth noting that both our ranking method and degree centrality work regardless of the time of attacks. We compute the degree centrality ranking of IOCs based on the fact that the node with a higher degree centrality is more important than a node with a lower one. For instance, if the degree centrality of a vulnerability is higher, it indicates that this vulnerability is exploited by more attackers or it affects more devices. The ranking result of degree centrality shown in Table 5 is consistent with the ranking result based on the proposed threat intelligence computing framework, demonstrating the capability of the CTI computing framework in ranking the importance of different types of IOCs.

6.2 Attack Preference Modeling

Attack preference modeling is meaningful for security organizations to gain insight into the attack intention of attackers, build attack portraits, and develop personalized defense strategies. Here, we leverage HINTI to integrate different types of IOCs and their interdependent relationships to comprehensively depict the picture of attack events, which helps model the attack preferences. With the proposed threat intelligence computing framework, we model attack preferences by clustering the embedded attackers’ vectors.

In this task, each malicious IP address is treated as an intruder, and its attack preferences are mainly reflected in three features including the platforms it destroys (including *Windows*, *Linux*, *Unix*, *ASP*, *Android*, *Apache*, etc), the industries it invades (e.g., *education*, *finance*, *government*, *Internet of Things*, and *Industrial control system*, etc), and the exploit types it employs (e.g., *DOS*, *Buffer overflow*, *Execute code*,

Table 5: The significance ranking of different types of IOCs. (*CVE1* : CVE-2017-0146, *CVE2* : CVE-2006-5911, *CVE3* : CVE-2008-6543, *CVE4* : CVE-2012-1199, *CVE4* : CVE-2006-4985; AR: Authoritative Ranking, DC: Degree Centrality value.)

Vulnerability			Attacker			Platform			Attack Type		
No.	AR	DC	Monicker	AR	DC	Category	AR	DC	Exploit_type	AR	DC
<i>CVE1</i>	0.2713	7,643	Meatsploit	0.2764	549	PHP	0.4562	17,865	Webapps	0.5494	11,648
<i>CVE2</i>	0.2431	7,124	GSR team	0.1391	327	windows	0.2242	13,793	DOS	0.1772	8,741
<i>CVE3</i>	0.2132	6,833	Ihsan	0.0698	279	Linux	0.0736	8,792	Overflow	0.1533	7,652
<i>CVE4</i>	0.1826	6,145	Techsa	0.0695	247	Linux86	0.0623	8,147	CSRF	0.0966	5,433
<i>CVE5</i>	0.1739	5,637	Aurimma	0.0622	204	ASP	0.0382	5,027	SQL	0.0251	2,171

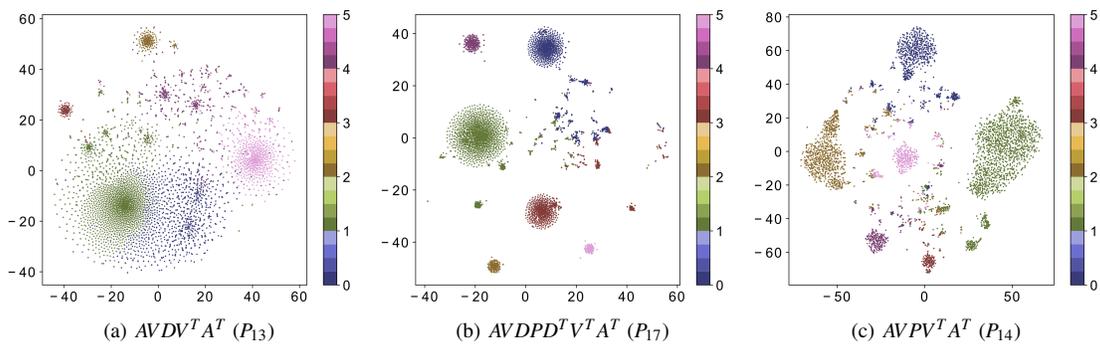


Figure 8: The performance of attack preference modeling with different meta-paths, in which the preference of attacker i is reduced to a two-dimensional space (x_i, y_i) and each cluster represents a group with a specific attack preference.

Sql injection, XSS, Gain information, etc).

Specifically, we first utilize our proposed threat intelligence computing framework to embed each attacker into a low-dimensional vector space, and then perform *DBSCAN* algorithm on the embedded vector to cluster attackers with the same preferences into corresponding groups. Figure 8 shows the top 3 clustering results under different types of meta-paths, in which the meta-path $AVDPD^T V^T A^T$ (P_{17}) performs the best performance with compact and well-separated clusters, indicating that it contains richer semantic relationships for characterizing attack preferences than other meta-paths.

To verify the effectiveness of attack preference modeling, we identify 5,297 distinct attackers (each unique IP address is treated as an attacker) who have submitted at least 10 cyber attacks. For these attackers, five cybersecurity researchers consisting of three doctoral and two master students spent about fortnight to manually annotate their attack preferences from three perspectives: the platforms they destroyed, the industries they attacked, and the attack types they exploited. To ensure the accuracy of data labeling, we test the consistency of the tags for the 5,297 attackers and remove the samples with ambiguous tags. As a result, we obtain 3,000 samples with consistent tags. Based on the labeled samples, we further evaluate the performance of different meta-paths on model-

Table 6: Performance of modeling attack preference with different meta-paths.

Metapath	Accuracy	Precision	Micro-F1
P_1	74.31	76.22	75.25
P_4	71.16	73.27	72.16
P_5	69.15	71.43	70.27
P_{12}	72.14	76.46	74.24
P_{13}	79.65	81.31	80.47
P_{14}	77.48	79.34	78.40
P_{15}	80.17	79.76	79.96
P_{17}	81.39	81.72	81.55

ing attack preferences. In the attack modeling scenario, we only focus on the meta-paths that both the start node and the end node are attackers. The experimental results are demonstrated in Table 6. Obviously, different meta-paths display different abilities in characterizing the attack preferences of cyber intruders. The performance when using P_{17} is more

superior than the one with other meta-paths, which indicates that P_{17} holds more valuable information that characterizes the attack preferences of cybercriminals, since P_{17} includes the semantics information of P_1, P_4, P_5 and $P_{12} \sim P_{15}$.

In addition, we compare the capabilities of our proposed computing framework with those of other state-of-the-art embedding methods in terms of attack preference modeling. Our analysis result shows that the accuracy of attack preference modeling reaches 0.81, which outperforms the existing popular models *Node2vec* (with precision of 0.71) [1], *metapath2vec* (with precision of 0.73) [11] and *HAN* (with precision of 0.76) [42]. The performance improvement can be attributed to the following characteristics. First, our computing framework utilizes weight-learning to learn the significance of different meta-paths for evaluating the similarity between attackers. Second, the proposed computing framework leverages GCN to learn the structural information between attackers to obtain more discriminative structural features that improves the performance of attack preference modeling.

6.3 Vulnerability Similarity Analysis

Vulnerability classification or clustering is crucial for conducting vulnerability trend analysis, the correlation analysis of incidents and exploits, and the evaluation of countermeasures. The traditional vulnerability analysis relies on the manual investigation of the source codes, which requires expert expertise and consumes considerable efforts. In this section, we propose an unsupervised vulnerability similarity analysis method based on the proposed threat intelligence computing framework, which can automatically group similar vulnerabilities into corresponding communities. Particularly, the vulnerability-related IOCs are first embedded into a low-dimensional vector space using CTI computing framework. Then, the *DBSCAN* algorithm is performed on the embedded vector space to cluster vulnerabilities into corresponding communities. The clustering results are presented in Figure 9.

Figure 9 (c) shows all vulnerabilities are clustered into 12 clusters using meta-path $VDPD^T V^T (P_{16})$, which is very close to the classification standard (i.e., 13) recommended by *CVE Details*, an authoritative database that publishes vulnerability information. By manually analyzing the training samples, we find that *HTTP Response Splitting* vulnerability does not appear in our dataset. Therefore, our cluster number (i.e., 12) is consistent with *CVE Details*¹¹. To further validate the effectiveness of threat intelligence computing framework for vulnerability clustering, we randomly select 100 vulnerabilities from each cluster for manual inspection to measure the consistency of the vulnerability types in each cluster, and the results are presented in Table 7. Obviously, the clustering performance of cluster 8 (i.e., File Inclusion) and cluster 10 (i.e., Directory Traversal) is remarkably worse than other clusters. To explain the reason, we examine our training data and the

¹¹<https://www.cvedetails.com/>

Table 7: Accuracy of vulnerability clustering.

Cluster ID	Vulnerability type	Accuracy
cluster1	Denial of Service	80.12
cluster2	XSS	83.53
cluster3	Execute Code	81.50
cluster4	Overflow	76.50
cluster5	Gain Privilege	91.56
cluster6	Bypass Something	71.74
cluster7	CSRF	93.27
cluster8	File Inclusion	61.72
cluster9	Gain Informa	70.42
cluster10	Directory Traversal	69.49
cluster11	Memory Corruption	81.56
cluster12	SQL Injection	80.67
average	#	78.51

computing framework. We found that the proportion of these two types of vulnerabilities is too small (cluster 8 is 3.4% and cluster 10 is 4.2%), making our computing framework very likely to be under-fit with insufficient data. However, the proposed computing framework performs well on most types of vulnerabilities in an unsupervised manner, especially given sufficient samples (e.g., cluster 7 is 17.6% and cluster 5 is 15.7%).

In addition, by examining the clustering results, we have an observation that the vulnerabilities in the same cluster are likely to have evolutionary relationships. For instance, *CVE-2018-0802*, an office zero-day vulnerability, is evolved from the *CVE-2017-11882*. They both include *EQNEDT32.exe* file used to edit the formula in Office software, which allows remote attackers to execute arbitrary codes by constructing a malformed font name. The modeling and computation of interdependent relationships among IOCs in HINTI facilitate the discovery of such intricate connections between vulnerabilities.

In summary, HINTI is capable of depicting a more comprehensive threat landscape, and the proposed CTI computing framework has the ability to bring novel security insights toward different real-world security applications. However, there are still numerous opportunities for enhancing these security applications. Specifically, for attack preference modeling, although each individual IP address is treated as an attacker, we cannot determine whether it belongs to a real attacker or is disguised by a proxy. Fortunately, even if the real attack address cannot be captured, understanding the attack preferences of these IP proxies, which are widely used in cybercrime, is also meaningful for gaining insight into the cyber threats. For vulnerability similarity analysis, data imbal-

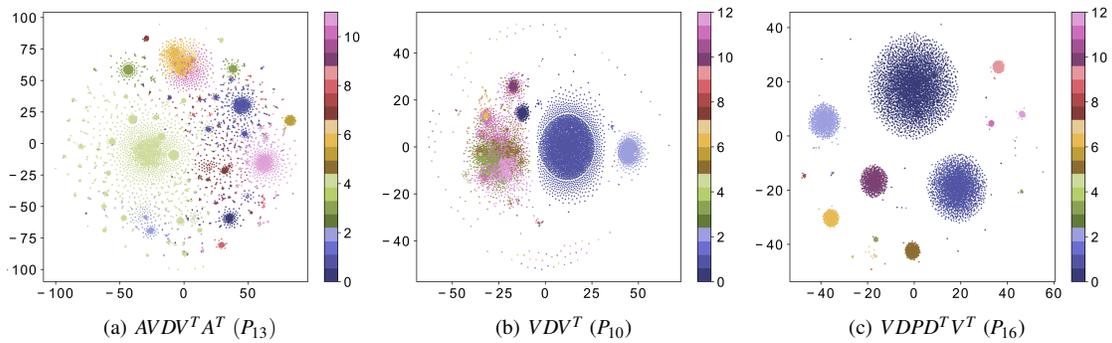


Figure 9: Illustration of the vulnerability similarity analysis based on different meta-paths, in which vulnerability i can be reduced into a two-dimensional space (x_i, y_i) and each cluster indicates a particular type of vulnerability.

ance issue affects the performance of model, and inadequate training samples often result in model underfitting, as shown in the case of cluster 8 and cluster 10.

7 Related Work

Cyber Treat Intelligence. An increasing number of security vendors and researchers start exploring CTI for protecting system security and defending against new threat vectors [28]. Existing CTI extraction tools such as *IBM X-Force*¹², *Threatcrowd*¹³, *Opencti.io*¹⁴, *AlienVault*¹⁵, *CleanMX*¹⁶ and *PhishTank*¹⁷ use regular expression to synthesize IOC from the descriptive texts. However, these methods often produce high false positive rate by misjudging legitimate entities as IOCs [22].

Recently, Balzarotti et al. [2] develop a system to extract IOCs from web pages and identify malicious URLs from JavaScript codes. Sabottke et al. [31] propose to detect potential vulnerability exploits by extracting and analyzing the tweets that contain “CVE” keyword. Liao et al. [22] present a tool, *iACE*, for automatically extracting IOCs, which excels at processing technology articles. Nevertheless, *iACE* identifies IOCs from a single article, which does not consider the rich semantic information from multi-source texts. Zhao et al. [46] define different ontologies to describe the relationship between entities based on expert knowledge. Numerous popular CTI platforms including *IODEF* [9], *STIX* [3], *TAXII* [40], *OpenIOC* [13], and *CyBox* [19] focus on extracting and sharing CTI. Yet, none of the existing approaches could uncover the interdependent relations among CTIs extracted from multi-source texts, let alone quantifying CTIs’ relevance and mining valuable threat intelligence hidden behind the isolated CTIs.

¹²<https://exchange.xforce.ibmcloud.com/>

¹³<https://www.threatcrowd.org/>

¹⁴<https://demo.opencti.io/>

¹⁵ <https://otx.alienvault.com/>

¹⁶<http://list.clean-mx.com>

¹⁷<https://www.phishtank.com>

Heterogeneous Information Network. Real-world systems often contain a large number of interacting, multi-typed objects, which can naturally be expressed as a heterogeneous information network (HIN). HIN, as a conceptual graph representation, can effectively fuse information and exploit richer semantics in interacting objects and links [37]. HIN has been applied to network traffic analysis [38], public social media data analysis [45], and large-scale document analysis [41]. Recent applications of HIN include mobile malware detection [14] and opioid user identification [12]. In this paper, *for the first time*, we use HIN for CTI modeling.

Graph Convolutional Network. Graph convolutional networks (GCN) [17] has become an effective tool for addressing the task of machine learning on graphs, such as semi-supervised node classification [17], event classification [29], clustering [8], link prediction [27], and recommended system [44]. Given a graph, GCN can directly conduct the convolutional operation on the graph to learn the nonlinear embedding of nodes. In our work, to discern and reveal the interactive relationships between IOCs, we utilize GCN to learn more discriminative representation from attributes and graph structure simultaneously, which is the premise for threat intelligence computing.

8 Discussion

Data Availability. The proposed framework assumes that sufficient threat description data can be obtained for generating comprehensive and the latest CTIs. Fortunately, with the growing prosperity of social media, an increasing number of security-related data (e.g., blogs, posts, news and open security databases) can be collected effortlessly. To automatically collect security-related data, we develop a crawler system to collect threat description data from 73 international security sources (e.g., blogs, hacker forum posts, security bulletins, etc), providing sufficient raw materials for generating cyber threat intelligence.

Model Extensibility. In this paper, 6 types of IOCs and 9

types of relationships are modeled in HINTI. However, our proposed framework is extensible, in which more types of IOCs and relationships can be enrolled to represent richer and more comprehensive threat information, such as malicious domains, phishing Emails, attack tools, their interactions, etc. **High-level Semantic Relations.** In view of the computational complexity of the model, our threat intelligence computing method focuses on utilizing the meta-paths to quantify the similarity between IOCs while ignoring the influence of the meta-graph on it, which inevitably misses characterizing some high-level semantic information. Nevertheless, the proposed computing framework introduces an attention mechanism to learn the signification of different meta-paths to characterize IOCs and their interactive relationships, which effectively compensates for the performance degradation caused by ignoring the meta-graphs.

Security Knowledge Reasoning. Although our proposed framework exhibits promising results in CTI extraction and modeling computing, how to implement advanced security knowledge reasoning and prediction is still an open problem, e.g., it remains challenging to predict whether a vulnerability could potentially affect a particular type of devices in the future. We will investigate this problem in the future.

9 Conclusion

This work explores a new direction of threat intelligence computing, which aims to uncover new knowledge in the relationships among different threat vectors. We propose HINTI, a cyber threat intelligence framework, to model and quantify the interdependent relationships among different types of IOCs by leveraging heterogeneous graph convolutional networks. We develop a multi-granular attention mechanism to learn the importance of different features, and model the interdependent relationships among IOCs using HIN. We propose the concept of threat intelligence computing and present a general intelligence computing framework based on graph convolutional networks. Experimental results demonstrate that the proposed multi-granular attention based IOC extraction method outperforms the existing state-of-the-art methods. The proposed threat intelligence computing framework can effectively mine security knowledge hidden in the interdependent relationships among IOCs, which enables crucial threat intelligence applications such as threat profiling and ranking, attack preference modeling, and vulnerability similarity analysis. We would like to emphasize that the knowledge discovery among interdependent CTIs is a new field that calls for a collaborative effort from security experts and data scientists.

In future, we plan to develop a predicative and reasoning model based on HINTI and explore preventative countermeasures to protect cyber infrastructure from future threats. We also plan to add more types of IOCs and relations to depict a more comprehensive threat landscape. Moreover, we will leverage both meta-paths and meta-graphs to characterize the

IOCs and their interactions to further improve the embedding performance, and to strike a balance between the accuracy and computational complexity of the model. We will also investigate the feasibility of security knowledge prediction based on HINTI to infer the potential future relationships between the vulnerabilities and devices.

Acknowledgement

We would like to thank our shepherd Tobias Fiebig, and the anonymous reviewers for providing valuable feedback on our work. We also thank Hao Peng and Lichao Sun for their feedback on the early version of this work. This work was supported in part by National Science Foundation grants CNS1950171, CNS-1949753. It was also supported by the NSFC for Innovative Research Group Science Fund Project (62141003), National Key R&D Program China (2018YFB0803 503), the 2018 joint Research Foundation of Ministry of Education, China Mobile (MCM20180507) and the Opening Project of Shanghai Trusted Industrial Control Platform (TICPSH202003020-ZC). Any opinions, findings, and conclusions or recommendations expressed in this material do not necessarily reflect the views of any funding agencies.

References

- [1] Jure Leskovec Aditya Grover. node2vec: Scalable feature learning for networks. In *Acm Sigkdd International Conference on Knowledge Discovery Data Mining*, 2016.
- [2] Marco Balduzzi, Marco Balduzzi, and Davide Balzarotti. Automatic extraction of indicators of compromise for web applications. In *WWW*, 2016.
- [3] Sean Barnum. Standardizing cyber threat intelligence information with the structured threat information expression (stix). *Mitre Corporation*, 11:1–22, 2012.
- [4] Eric W Burger, Michael D Goodman, Panos Kampanakis, and Kevin A Zhu. Taxonomy model for cyber threat intelligence information exchange technologies. In *Proceedings of the 2014 ACM Workshop on Information Sharing & Collaborative Security*, pages 51–60, 2014.
- [5] Onur Catakoglu, Marco Balduzzi, and Davide Balzarotti. Automatic extraction of indicators of compromise for web applications. *The web conference*, pages 333–343, 2016.
- [6] Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks. pages 740–750, 2014.

- [7] Qian Chen and Robert A. Bridges. Automated behavioral analysis of malware a case study of wannacry ransomware. In *16th IEEE ICMLA*, pages 454–460, 2017.
- [8] Weilin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Chojui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. *knowledge discovery and data mining*, pages 257–266, 2019.
- [9] Roman Danyliw, Jan Meijer, and Yuri Demchenko. The incident object description exchange format. *International Journal of High Performance Computing Applications*, 5070:1–92, 2007.
- [10] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *23rd ACM SIGKDD*, pages 135–144, 2017.
- [11] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. metapath2vec: Scalable representation learning for heterogeneous networks. In *23rd ACM SIGKDD*, pages 135–144. ACM, 2017.
- [12] Yujie Fan, Yiming Zhang, Yanfang Ye, and Xin Li. Automatic opioid user detection from twitter: Transductive ensemble built on different meta-graph based similarities over heterogeneous information network. In *IJCAI*, pages 3357–3363, 2018.
- [13] Fireeye. Openioc. <https://www.fireeye.com/blog/threat-research/2013/10/openioc-basics.html>. Accessed January 20, 2020.
- [14] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1507–1515, 2017.
- [15] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv:1508.01991*, 2015.
- [16] Michael D Iannacone, Shawn Bohn, Grant Nakamura, John Gerth, Kelly MT Huffer, Robert A Bridges, Erik M Ferragut, and John R Goodall. Developing an ontology for cyber security knowledge graphs. *CISR*, 15:12, 2015.
- [17] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv: Learning*, 2016.
- [18] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [19] Tero Kokkonen. Architecture for the cyber security situational awareness system. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 294–302. Springer, 2016.
- [20] Mehmet Necip Kurt, Yasin Yılmaz, and Xiaodong Wang. Distributed quickest detection of cyber-attacks in smart grid. *IEEE Transactions on Information Forensics and Security*, 13(8):2015–2030, 2018.
- [21] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *ICML*, pages 282–289, 2001.
- [22] Xiaojing Liao, Yuan Kan, Xiao Feng Wang, Li Zhou, and Raheem Beyah. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *ACM Sigsac Conference on Computer Communications Security*, 2016.
- [23] Rob Mcmillan. Open threat intelligence. <http://www.gartner.com/doc/2487216/definition-threat-intelligence>. Accessed January 20, 2020.
- [24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [25] Sudip Mittal, Prajit Kumar Das, Varish Mulwad, Anupam Joshi, and Tim Finin. Cybertwitter: Using twitter to generate alerts for cybersecurity threats and vulnerabilities. In *Proceedings of the 2016 IEEE Advances in Social Networks Analysis and Mining*, pages 860–867, 2016.
- [26] Eric Nunes, Ahmad Diab, Andrew Gunn, Ericsson Marin, Vineet Mishra, Vivin Paliath, John Robertson, Jana Shakarian, Amanda Thart, and Paulo Shakarian. Darknet and deepnet mining for proactive cybersecurity threat intelligence. In *2016 IEEE ISI*, pages 7–12, 2016.
- [27] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding. pages 2609–2615, 2018.
- [28] P Pawlinski, P Jaroszewski, P Kijewski, L Siewierski, P Jacewicz, P Zielony, and R Zuber. Actionable information for security incident response. *European Union Agency for Network and Information Security, Heraklion, Greece*, 2014.

- [29] Hao Peng, Jianxin Li, Qiran Gong, Yangqiu Song, Yuanxing Ning, Kunfeng Lai, and Philip S Yu. Fine-grained event categorization with heterogeneous graph convolutional networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3238–3245. AAAI Press, 2019.
- [30] Sara Qamar, Zahid Anwar, Mohammad Ashiqur Rahman, Ehab Al-Shaer, and Bei-Tseng Chu. Data-driven analytics for cyber-threat intelligence and information sharing. *Computers & Security*, 67:35–58, 2017.
- [31] Carl Sabottke, Octavian Suci, and Tudor Dumitras. Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits. In *USENIX Security*, 2015.
- [32] Carl Sabottke, Octavian Suci, and Tudor Dumitras. Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits. In *24th USENIX Security*, pages 1041–1056, 2015.
- [33] Deepak Sharma and Avadhesh Surolia. *Degree Centrality*. Springer New York, 2013.
- [34] Saurabh Singh, Pradip Kumar Sharma, Seo Yeon Moon, Daesung Moon, and Jong Hyuk Park. A comprehensive study on apt attacks and countermeasures for future networks and communications: challenges and solutions. *Journal of Supercomputing*, pages 1–32, 2016.
- [35] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: Principles and methodologies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 3(2):1–159, 2012.
- [36] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: a structural analysis approach. *ACM SIGKDD*, 14(2):20–28, 2013.
- [37] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment*, 4(11):992–1003, 2011.
- [38] Yizhou Sun, Brandon Norick, Jiawei Han, Xifeng Yan, Philip S Yu, and Xiao Yu. Pathselclus: Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. *ACM Transactions on TKDD*, 7(3):11, 2013.
- [39] Wiem Tounsi and Helmi Rais. A survey on technical threat intelligence in the age of sophisticated cyber attacks. *Computers & Security*, 72:212–233, 2018.
- [40] Thomas D Wagner, Esther Palomar, Khaled Mahbub, and Ali E Abdallah. Towards an anonymity supported platform for shared cyber threat intelligence. *risks and security of internet and systems*, pages 175–183, 2017.
- [41] Chenguang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. Text classification with heterogeneous information network kernels. In *13th AAAI*, 2016.
- [42] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Peng Cui, P. Yu, and Yanfang Ye. Heterogeneous graph attention network. 2019.
- [43] Jie Yang, Shuailong Liang, and Yue Zhang. Design challenges and misconceptions in neural sequence labeling. *arXiv: Computation and Language*, 2018.
- [44] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018.
- [45] Jiawei Zhang, Xiangnan Kong, and Philip S Yu. Transferring heterogeneous links across location-based social networks. In *The 7th ACM international conference on Web search and data mining*, pages 303–312, 2014.
- [46] Yishuai Zhao, Bo Lang, and Ming Liu. Ontology-based unified model for heterogeneous threat intelligence integration and sharing. In *2017 11th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 11–15, 2017.

Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI

Benjamin Bowman Craig Laprade Yuede Ji H. Howie Huang
Graph Computing Lab
George Washington University

Abstract

In this paper we present a technique for detecting lateral movement of Advanced Persistent Threats inside enterprise-level computer networks using unsupervised graph learning. Our detection technique utilizes information derived from industry standard logging practices, rendering it immediately deployable to real-world enterprise networks. Importantly, this technique is fully unsupervised, not requiring any labeled training data, making it highly generalizable to different environments. The approach consists of two core components: an authentication graph, and an unsupervised graph-based machine learning pipeline which learns latent representations of the authenticating entities, and subsequently performs anomaly detection by identifying low-probability authentication events via a learned logistic regression link predictor. We apply this technique to authentication data derived from two contrasting data sources: a small-scale simulated environment, and a large-scale real-world environment. We are able to detect malicious authentication events associated with lateral movement with a true positive rate of 85% and false positive rate of 0.9%, compared to 72% and 4.4% by traditional rule-based heuristics and non-graph anomaly detection algorithms. In addition, we have designed several filters to further reduce the false positive rate by nearly 40%, while reducing true positives by less than 1%.

1 Introduction

According to the 2019 FireEye M-Trends report [5], the median time to detection of a network intrusion was 78 days. While this is an impressive improvement from the 418 days reported in 2011, this still means an adversary would have over 2 months inside an environment to accomplish their mission prior to detection. Additionally, nearly half of all compromises are detected via external sources, indicating that the tools currently employed by enterprise-level cyber defenders are insufficient for detecting the highly sophisticated modern-day adversaries.

Existing systems and techniques for detecting network intrusions rely heavily on signatures of known-bad events [25], such as file hashes of malware, or byte streams of malicious network traffic. While these techniques are able to detect relatively unskilled adversaries who use known malware and common exploitation frameworks, they provide almost no utility for detecting advanced adversaries, coined Advanced Persistent Threats (APTs), who will use zero-day exploits, novel malware, and stealthy procedures.

Similarly, the state-of-the-art behavioral analytics [26] in use today by network defenders utilize relatively rudimentary statistical features such as the number of bytes sent over a specific port, number of packets, ratio of TCP flags, etc. Not only are these types of analytics relatively noisy in terms of false positives, but they are also challenging to investigate due to their limited information and scope. For example, the fact that a particular host sent 50% more network packets in a given day could be indicative of many different events, ranging from data exfiltration, botnet command & control, to a myriad of other possibilities, most of which would not indicate a compromise, such as streaming a video.

To address these challenges, our approach is to build an abstract, behavior-based, graph data model, with key elements related to the particular behavior of interest we are trying to detect. Specifically, we model a computer network using a graph of authenticating entities, and the target behavior we detect is anomalous authentication between entities indicative of lateral movement within the network. Lateral movement is a key stage of APT campaigns when an attacker will authenticate to new resources and traverse through the network in order to gain access to systems and credentials necessary to carry out their mission [17, 21]. This is very challenging to detect as attackers will often use legitimate authentication channels with valid credentials as opposed to noisy exploitation procedures.

In order to effectively detect lateral movement, we first convert our input data, which is in the form of industry standard authentication logs, into a representation which will allow for not only learning about individual authentication events, but

also the authentication behavior of the network as a whole. To that end, we construct an authentication graph, where nodes represent authenticating entities which can either be machines or users, and edges represent authentication events. Next, we utilize an unsupervised node embedding technique where latent representations are generated for each vertex in the graph. Finally, we train a link predictor algorithm on these vertex embeddings, and utilize this link predictor to identify low-probability links in new authentication events.

We apply our technique on two distinct datasets representing two contrasting computer networks. The PicoDomain dataset is a small simulated environment we developed in-house with only a few hosts, and spanning only 3 days. The second dataset is from Los Alamos National Labs (LANL) [11] and is a real-world network capture from their internal enterprise computer network spanning 58 days with over 12,000 users and 17,000 computers. In both cases, there is labeled malicious authentication events associated with APT-style activity which were used as ground truth for evaluation purposes. We were able to detect the malicious authentication events in the real-world dataset with a true positive rate of 85% and a false positive rate of only 0.9%. In comparison, traditional heuristics, and non-graph based machine learning methods, were able to achieve at best 72% true positive rate and 4.4% false positive rate. Understanding that modern day cyber defenders are frequently receiving far too many false positives, we spent additional time building simple filters that allowed us to further reduce our false-positive rate by nearly 40% on the LANL dataset, while reducing true positives by less than 1%.

In summary, our contributions of this work are as follows:

- A graph data structure for modeling authentication behavior within enterprise-level computer networks based on information available in industry standard log files.
- An unsupervised graph-learning technique for identifying anomalous authentication events which are highly indicative of malicious lateral movement.
- Experiments on two datasets showing the strength of graph learning for this application domain.

The remaining of this paper will be laid out as follows. Section 2 will provide some background into authentication protocols, the graph structure, and define the problem of lateral movement. Section 3 will discuss our proposed method and explain the learning algorithm. Section 4 will discuss our experimental evaluation and results. Section 5 will discuss the related work. Section 6 will discuss some limitations of our approach and our planned future work, and Section 7 will conclude.

2 Background & Problem Definition

In this section we will discuss some background on authentication in enterprise networks, how we build our graph structure, and define the problem of lateral movement.

2.1 Authentication

Modern enterprise computer networks rely on the ability to manage the permissions and privileges of users in order to maintain a safe and secure network. Users in the enterprise network will be given explicit permissions to access resources within the environment ranging from folders and network share drives, to applications and services. To make this possible, there have been many network authentication protocols developed through the years, which allow users to authenticate to resources in the network in order to verify that they have the privileges necessary to perform a certain action.

Common authentication protocols in today's enterprise computer networks include protocols such as Kerberos, NTLM, SAML, and others. Each one is designed to be a secure way to authenticate users inside an environment, and each has the ability to be abused. APT-level adversaries are well-versed in the workings of these authentication protocols, and they are often abused during an attack campaign. For example, the well-known "Pass the Hash" attack is a weakness in the NTLM implementation where the hash of a user's password, which can often be harvested from system memory, is used to authenticate to additional resources by the attacker.

Because hackers often abuse existing authentication channels, logs related to these critical protocols are valuable to the security analyst and detection algorithms. Typically these logs capture key information such as the account that is requesting to authenticate, the origin of the request, what they are attempting to authenticate to, as well as the result of that authentication request. Additionally, as authentication in the environment is network activity, we have the ability to capture this critical information from centralized network taps, rather than requiring expensive host-based log collection.

2.2 Graph Structure

There were two main considerations in how we chose to build our graph data structure. First, we wanted the input data to be highly accessible to our network defenders. This means utilizing data that is likely already being collected at the enterprise scale. While some smaller enterprises may have the luxury of collecting verbose system logs from all endpoints, larger enterprises are limited to coarse feeds from centralized resources such as network sensors or domain controllers. Second, we wanted the data to provide clear and concise information related to our target detection of lateral movement. Therefore, we design our algorithm to utilize network-level authentication logs generated from Zeek sensors [31] (formerly

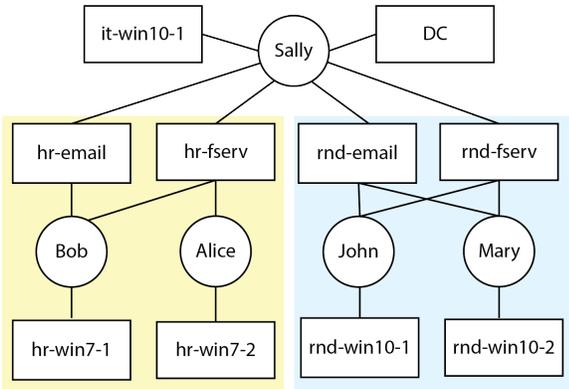


Figure 1: Example of an authentication graph for a small simulated network.

Bro). Specifically, we utilize the Kerberos logging capability, which generates protocol specific logging on the Kerberos authentication protocol which is utilized in the majority of Microsoft Windows domains. The technique is easily adaptable, however, to other authentication logs such as host-based authentication logs, NTLM logs, Active Directory logs, or others, providing they can uniquely identify authentication events between user and system identities in the network.

For Kerberos logs, we extract the client and service principals, which are unique identifiers associated with users and services in the network, as well as the source IP address of the requesting entity, which will uniquely identify the machine from which the client is operating. The destination IP address will always be the IP of the Kerberos server itself, and thus does not add valuable information to our graph. Here is an example of content we extract from the Kerberos logs with their respective Zeek column headings:

client	id_orig_h	service
jdoe/G.LAB	10.1.1.152	host/hr-1.g.lab

This record shows that the user *jdoe* of domain *G.LAB* authenticated to service *host/hr-1.g.lab*, which is a host in the network, from IP address *10.1.1.152*.

Definition 1 An *authentication graph (AG)* is defined as a graph $G = (V, E)$ with a node type mapping $\phi: V \rightarrow A$ and an edge type mapping $\psi: E \rightarrow R$, where V denotes the node set and E denotes the edge set, $A = \{IP, user, service\}$ and $R = \{authentication\}$.

A simple authentication graph generated from a small simulated computer network is shown in Figure 1. We can infer from this graph that there are two separate organizational units in our enterprise: the *hr* unit and the *rnd* unit, each with two user nodes (*Bob* and *Alice*, *John* and *Mary*) interacting with user workstations represented as service nodes (*hr-win7-1*, *hr-win7-2*, *rnd-win10-1*, *rnd-win10-2*), as well as some email

servers and file servers (*hr-email*, *hr-fserv*, *rnd-email*, *rnd-fserv*). We can see that user *Sally* is a network administrator, as she has authentication activity to the Domain Controller service node (*DC*) in the environment, the email and file server nodes, as well as her own workstation node (*it-win10-1*). Note that for display purposes, the IP nodes have been collapsed into their representative service nodes.

2.3 Lateral Movement

Lateral movement is a key stage of APT-level attack campaigns as seen in various attack taxonomies such as the Lockheed Martin Cyber Kill Chain [17], and the MITRE ATT@CK framework [21]. Figure 2 provides a simplified version of an APT-style campaign. After some initial compromise, and prior to domain ownership by the adversary, there is a cycle of lateral movement through the network. In most cases, the system that is initially compromised will be a low privileged account, typically a user workstation. This is due to the prevalence of client-side attacks (e.g., phishing), which are much more effective on typical, low-privilege users, as opposed to high-privilege IT professionals. Thus, the attacker almost always gains a foothold on a low privilege system and is thus required to move laterally through the network to achieve their goals.

Definition 2 *Lateral movement* is defined as a malicious path $\langle u, v \rangle$ conducted by an attacker in an organization's network characterized by the authentication graph, where u, v belong to entity set $\{IP, user, service\}$.

For example, in Figure 1, if the user *Alice* fell victim to a phishing email and downloaded malware, the attacker would gain their initial foothold as account *Alice* on *hr-win7-2*. As *Alice* is a low-privilege account, it is unlikely that the attacker would be able to do much harm to the enterprise at large, such as installing ransomware on all the systems in the network, or exfiltrating highly sensitive business data. Therefore, the attacker would be required to move laterally to systems and

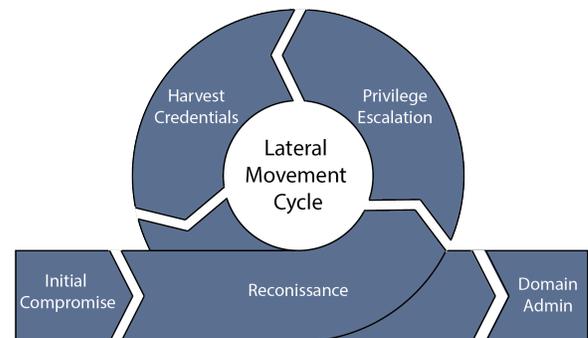


Figure 2: An APT-style campaign showing the cycle of lateral movement after initial compromise and prior to full domain ownership.

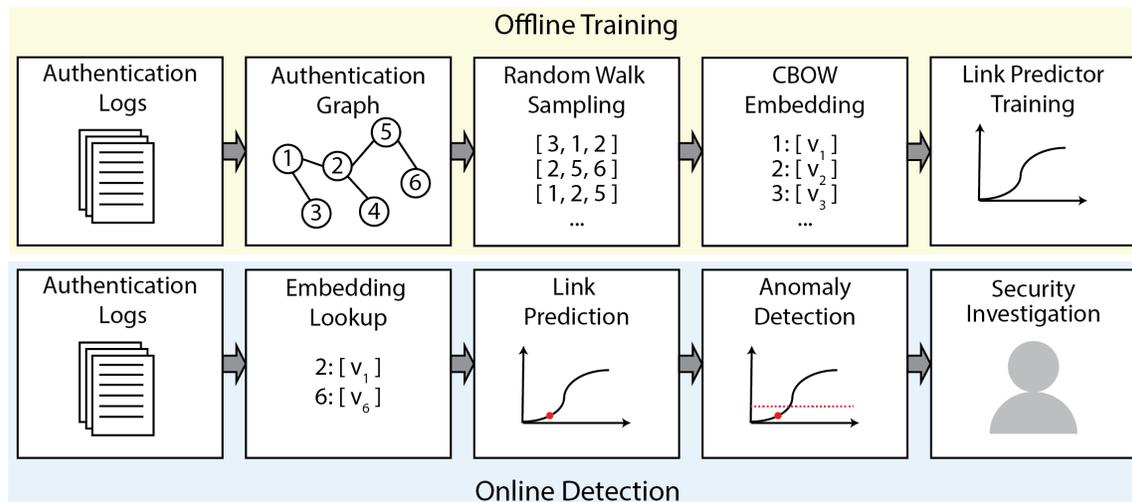


Figure 3: Full algorithm pipeline including offline training of node embeddings and logistic regression link predictor, as well as online detection via an embedding lookup, link prediction, and threshold-based anomaly detection.

accounts that have higher permissions in the environment. This can be done by exploitation of vulnerabilities, however, this is often a noisy and error prone process. More often, adversaries will harvest and abuse legitimate credentials from the set of compromised systems. In the case of our example, Alice could harvest the domain admin *Sally*'s credentials from the file server *hr-fserv* which *Sally* had previously authenticated to, and *Alice* has privileges to access. Now, with *Sally*'s credentials, *Alice* can authenticate from *hr-win7-2* to the *Domain Controller (DC)*. This attack could be characterized by the lateral movement path: $\langle hr-win7-2, Sally, DC \rangle$.

Existing techniques are not well suited to detect lateral movement within enterprise-scale environments. Most Intrusion Detection Systems (IDSs) are placed at the border of a network, and will fail to detect attacker actions after an initial foothold has been established. Even if the IDS had total visibility, an attacker using legitimate authentication channels would likely not trigger any alerts. Host-based security software relies almost exclusively on identifying signatures of known malware, and thus will prove ineffective at detecting APT-level adversaries who will move laterally through a network using novel malware or legitimate authentication mechanisms. Some environments may implement a Security Information Events Management (SIEM) System, which would allow for more complex log analytics. However, SIEMs are typically standard row or columnar data stores such as Splunk [26] which only allow for relatively basic statistical analysis of the data. Behavioral analytics implemented in SIEMs are typically simple aggregate trends of low level features such as bytes over particular ports and protocols.

3 Proposed Method

In this section we will discuss our proposed method for detecting lateral movement in enterprise computer networks. We will provide an overview of our machine learning pipeline, followed by detailed discussions of the node embedding process, the link predictor training, and the anomaly detection.

3.1 Overview

In order to detect lateral movement in enterprise computer networks, we generate authentication graphs as discussed previously and apply an unsupervised graph learning process to identify low probability links. Figure 3 shows the algorithm pipeline. During the offline training stage (the top half of the figure), we start by generating authentication graphs, then create node embeddings via a random walk sampling and embedding process, and finally train a logistic regression link predictor using the node embeddings and ground-truth edge information from the authentication graph.

During the online detection stage (the bottom half of the figure), new authentication events are processed resulting in new edges between authenticating entities. Embeddings for these entities are generated via an embedding lookup, and link prediction is performed using the trained logistic regression link predictor. Anomaly detection is performed via a (configurable) threshold value, where links below a particular probability threshold will be forwarded to security experts for investigation.

3.2 Node Embedding Generation

Node embedding generation is the process by which a d -dimensional vector is learned for each node in a graph. The

goal of these approaches is to generate a vector representation for each node which captures some degree of behavior within the network as a whole.

For the authentication graph, we use H to denote the set of node embeddings, $H = \{h_1, h_2, \dots, h_n\}$, where h_i denotes the node embedding for the i th node, and n denotes the number of nodes in the graph. In the beginning, nodes do not have embeddings, which means $h_i = \emptyset$.

In order to extract latent node representations from the graph, we utilize an unsupervised node embedding technique similar to *DeepWalk* [22], and *node2vec* [7]. We first sample our authentication graph via unbiased, fixed-length random walks. Specifically, for any node v in the graph, we will explore r random walks with a fixed-length l . For a random walk starting from node v , let v_i denote the i th node in the walk, the node sequence for this walk is generated with the following probability distribution:

$$P(v_i = x | v_{i-1} = y) = \begin{cases} \frac{1}{d_y}, & \text{if } (x, y) \in E \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where E denotes the edge set in the graph, and d_y is the degree of node y . This results in a set of random walk sequences $S = \{S_1, S_2, \dots, S_m\}$, where S_i denotes the i th random walk sequence, and m denotes the total number of sequences.

With the sequence set of the random walks, we then tune node embeddings via a Continuous-Bag-of-Words (CBOW) model with negative sampling as proposed in [18]. In the CBOW model, we predict the target node provided context nodes from the random walk sequence. We utilize negative sampling such that we only update the vectors of a subset of nodes that were not found in the particular context window of the target node.

We use the Noise Contrastive Estimation (NCE) loss as defined in Equation 2:

$$L = -[\log p(y = 1 | h_T, h_I) + \sum_{h_U \in N(h_I)} \log p(y = 0 | h_U, h_I)] \quad (2)$$

where y denotes the label, h_T denotes the embedding of the target node, h_I denotes the embedding of the input node which is the average of the context nodes, h_U denotes the embedding of a noise node, and $N(\cdot)$ denotes the set of noise node embeddings for that input. This loss function differentiates the target sample from noise samples using logistic regression [8].

Further, the probability for different labels of negative sampling is defined in Equation 3,

$$\begin{aligned} p(y = 1 | h_T, h_I) &= \sigma(h_T^T h_I) \\ p(y = 0 | h_T, h_I) &= \sigma(-h_T^T h_I) \end{aligned} \quad (3)$$

where $\sigma(\cdot)$ denotes the sigmoid function, and h_T^T denotes the column vector for h_T . Therefore, the final loss value is calculated by Equation 4.

$$L = -[\log \sigma(h_T^T h_I) + \sum_{h_U \in N(h_I)} \log \sigma(-h_T^T h_U)] \quad (4)$$

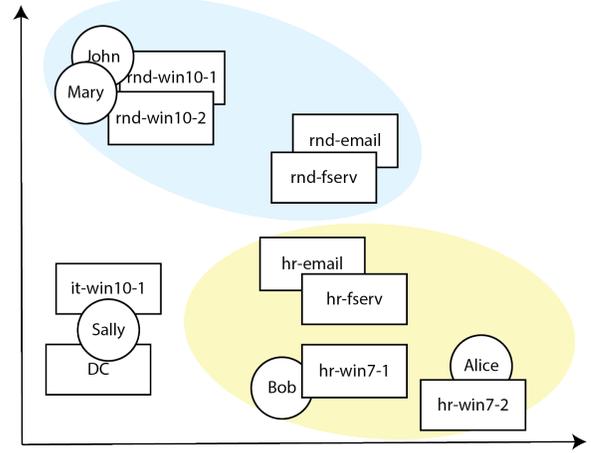


Figure 4: Example embedding space generated from a random-walk based node-embedding process.

By minimizing the loss value from Equation 4, we are able to tune our node embeddings such that we are more likely to predict our target node embedding h_T given the context node embeddings h_I , while simultaneously less likely to predict the negative sample node embeddings h_U given the same context h_I . We use Stochastic Gradient Descent (SGD) to minimize the loss function. In the end, we generate the output node embedding set $H' = \{h'_1, h'_2, \dots, h'_n\}$, where h'_i is the d -dimension embedding for node i .

In the context of the authentication graph, this process equates to predicting a user based on the machines and users found within at-most l -hops away. This will result in node embeddings where users who often authenticate to similar entities will be embedded in a similar region. Similarly, systems which share a user base will be found embedded in a similar region. This provides us the ability to then look at authentication events as events between two abstract vectors, as opposed to between distinct users and machines.

Figure 4 provides a 2-dimensional embedding space generated for the graph in Figure 1 using this node embedding process. We can see that the embedding of the graph corresponds nicely to the organizational units of the various users and systems. Additionally we see that the servers are clearly separated from the users and their workstations. Also, the network administrator is clearly separated from both organizational units. In addition, notice that the user *Alice* does not have an edge to the *hr-email* server in the authentication graph, despite clearly being a member of the *hr* organization. Even though this is the case, we can see that *Alice* is co-located in the embedding space with other *hr* users and systems. This fact will be crucial during the link prediction process, as even though there is no explicit link between *Alice* and the *hr-email* server, we would like our link prediction algorithm to predict a high probability for the authentication event between *Alice* and *hr-email*, considering it is perfectly

reasonable that *Alice* authenticates to the *hr-email* server.

3.3 Link Prediction

Next, we utilize a traditional logistic regression (LR) algorithm to provide us with a probability estimate that a particular authentication event occurs between two nodes a and b . Formally, our LR algorithm models:

$$P(y = 1|h') = \sigma(h') = \frac{1}{1 + e^{-w^\top h'}} \quad (5)$$

where y is the binary label indicating if an edge exists or not, the weight vector w contains the learned parameters, and h' is the element-wise multiplication of the node embeddings h_a and h_b defined in Equation 6, also known as the Hadamard product.

$$h_a \circ h_b = (h_a)_{ij} \cdot (h_b)_{ij} \quad (6)$$

We train the above model by generating a dataset of true and false edge embeddings from the ground truth authentication graph. The true edge set consists of all edges in the authentication graph:

$$E_T = h_a \circ h_b \forall (a, b) \in E \quad (7)$$

with each edge embedding receiving a binary label of 1. On the contrary, the false edge set consists of all edges that do not exist in the authentication graph:

$$E_F = h_a \circ h_b \forall (a, b) \notin E \quad (8)$$

with each edge embedding receiving a binary label of 0. Training on these two sets of data would cause significant over fitting as E_F contains every possible edge not in the original edge set E . Therefore, we down sample E_F via a random sampling process, and only train on the same number of false edges as found in E_T .

3.4 Anomaly Detection

Anomaly detection is achieved by applying our trained LR link predictor to new authentication events. First, authentication events are parsed into a set of edges between authenticating entities. Next, we perform an embedding lookup for the node embeddings generated during the training stage. The anomaly detection function A can be expressed as:

$$A(h_a, h_b) = \begin{cases} 1, & \text{if } f(h_a \circ h_b) < \delta \\ 0, & \text{otherwise} \end{cases} \quad (9)$$

where h_a and h_b are the embeddings for nodes a and b , and the function $f(\cdot)$ is the logistic regression link predictor trained on the true and false edges generated from our training graph. The parameter δ is the threshold for generating an alert. In this paper, we use a threshold of $\delta = 0.1$, or 10%, which we will show shortly yields good performance.

4 Evaluation

In this section we will evaluate our technique for detecting malicious authentication in enterprise networks. First we will discuss the datasets we used for evaluation, followed by a detailed description of the various methods we evaluated, and an analysis of our results. In an effort to further reduce false positives, we make some observations about the data and our results, and update our algorithm accordingly.

4.1 Datasets

We apply our malicious authentication detection to two datasets generated from contrasting computer networks. Table 1 provides details on each dataset. We discuss both datasets in detail below.

Table 1: Dataset Details

	PicoDomain	LANL
Duration in Days	3	58
Days with Attacks	2	18
Total Records	4686	1.05 B
Total Attack Records	129	749
User and Machine Accounts	86	99968
Computers	6	17666

PicoDomain is a dataset we generated in-house for cyber security research. It is designed to be a highly scaled-down environment which contains only the most critical elements commonly found in enterprise-level domains. Specifically, the PicoDomain consists of a small Windows-based environment with five workstations, a domain controller, a gateway firewall and router, and a small-scale internet that houses several web-sites as well as the adversarial infrastructure. A Zeek network sensor was installed inside the environment and placed such that it had visibility of traffic entering and leaving the network from the simulated Internet (north/south), as well as traffic between local systems in the simulated enterprise network (east/west). A total of three days of network traffic was captured. During this three day period, there was benign activity performed in a typical 9-5 workday pattern, such as browsing the web, checking e-mail, etc. Additionally, on days 2 and 3, we ran an APT-style attack campaign which included all stages of the kill chain. The attack campaign started with a malicious file downloaded from an e-mail attachment. This gave the attacker the initial foothold in the network. The attacker then proceeded to perform various malicious actions typically associated with APT-level campaigns. This included exploiting system vulnerabilities for privilege escalation, registry modifications to maintain persistence, credential harvesting via the tool Mimikatz, domain enumeration, and lateral movement to new systems via the legitimate Windows Management Instrumentation (WMI) service. At the end of the campaign,

the attacker was able to compromise a domain admin account, resulting in full network ownership by the attacker.

Comprehensive Cyber Security Events is a dataset released by Los Alamos National Labs (LANL) and consists of 58 consecutive days of anonymized network and host data [11]. There are over 1 billion events containing authentication activity for over 12,000 users and 17,000 computers in the network. An APT-style attack was performed during the data capture, and relevant authentication log entries were labeled as being malicious or benign. No further details were provided in the dataset as to what types of attacks were performed during the exercise. This is a limiting factor of this dataset, and, in fact, led to the generation of the previously mentioned PicoDomain dataset.

4.2 Methods Evaluated

We evaluate two variants of our proposed graph learning methods, as well as four different baseline techniques, which include two non-graph-based machine learning algorithms, as well as two traditional rule-based heuristics. We will discuss each below.

Graph Learning with Local View (GL-LV). This is our graph learning technique configured in such a way as to have a more localized view in our graph. This means our embeddings and link predictor will be optimized for nodes within a close proximity. To achieve this, we generate 20 random walks of length 10 for every node, and generate a 128-dimension embedding for each node based on a context window size of 2. This means each node will only consider a neighborhood of 2-hop neighbors in the embedding process. Our anomaly detection threshold is set at $\delta = 0.1$.

Graph Learning with Global View (GL-GV). This is our second graph learning variant which is very similar to the first, however this time configured to have a more global view of the graph. This means our embeddings and link predictor will be optimized for nodes that are further apart in our graph. To that end we used the same configuration as previously, however now setting the window size to 5. This means nodes will consider at most 5-hop neighbors during the embedding and link prediction process, which will give the algorithm a much broader view of the graph.

Local Outlier Factor (LOF) [2]. For a non-graph-based machine learning comparison, we implement the LOF anomaly detection algorithm. The LOF is a density-based anomaly detection approach, where relative local densities are compared between each sample, and those which are very different from their neighbors are considered anomalous. In order to generate features for this algorithm, we 1-hot encode the authentication events into an authentication vector containing a dimension for all authenticating entities. For each event, the dimensions corresponding to the various authenticating entities for that particular record will be set to 1, and all other dimensions will be 0. We then apply the LOF algorithm

to these vectors to identify anomalies.

Isolation Forest (IF) [15]. This is a second non-graph-based machine learning comparison technique. The Isolation Forest algorithm identifies samples that can be easily isolated from the dataset by simple decision trees as being anomalous. This is applied to the same authentication vectors as in the previous LOF method.

Unknown Authentication (UA). This is a more traditional rule-based heuristic which simply identifies all first-time authentication events as anomalous. During the training period, a list of known authentications is generated for each authenticating entity in the network. During the testing phase, any authentication event which was not seen during the training phase is considered as anomalous. After an anomalous result is generated the first time, the authentication event is added to the set of known authentications for the particular entity. This way we do not generate repeated anomalies for the same event.

Failed Login (FL). This is a second traditional rule-based heuristic which considers all failed login events as anomalous. As this technique does not require any training data, we only evaluate it on the test portion of the datasets.

4.3 Detection Analysis

Next we apply the six different algorithms discussed previously and evaluate their ability to detect malicious authentication in our two datasets. For all techniques, we report the number of true positives (TP), false positives (FP), as well as the true positive rate (TPR), and false positive rate (FPR).

PicoDomain. First we apply all techniques to the simulated PicoDomain dataset. We split the dataset into training and testing, with the training data consisting of authentication activity before the APT attack began, and the testing data containing all other activity. As this is a small dataset focused on malicious activity, the majority of the time period contains malicious events. As a result, there was only roughly 20% clean training data available. Thus our final train/test split on this data was about 20%/80%. For all 6 detection techniques, we only generate accuracy metrics on the testing dataset.

Table 2 shows the results for all six techniques. Not surprisingly, the UA detector performed very well, with 100% TPR, and only 1.5% FPR. This means all of the lateral movement associated with the APT campaign involved systems which did not have authentication activity during the training period, a characteristic that is likely only to hold in relatively small and simulated environments. We can also see that the failed login (FL) rule generated very few results, and only managed to detect a single event associated with the malicious activity. This is due to the fact that the APT campaign did not involve any brute-force password guessing attempts. The single failed login is likely due to user error during the attack.

Both ML techniques (LOF and IF) struggled to detect malicious events, with TPRs well below 50%, and FPRs as high

Table 2: Anomaly Detection Results on PicoDomain Dataset

Algorithm	TP	FP	TPR (%)	FPR (%)
UA	129	11	100	1.5
FL	1	15	0.8	2.0
LOF	41	19	32	2.5
IF	34	62	26	8.3
GL-LV	102	0	80	0.0
GL-GV	102	0	80	0.0

as 8.3%. This indicates that the pure authentication activity between entities, without the additional information present in the graph topology, is not sufficient for detecting lateral movement.

Our graph learning techniques, **GL-LV** and **GL-GV**, performed much better than the comparison ML techniques, achieving 80% TPR. This shows the strength of the graph topology for the detection of lateral movement. Additionally, the graph-learning approaches were able to reduce the FPR to 0% compared with the 1.5% of the **UA** detector. A low false positive rate is critical for anomaly detection techniques, as will be made clear by the next experiment on the LANL dataset. Interestingly, we see that the global view and local view had no effect on the performance. This again is likely due to the extremely small scale of this dataset. The average shortest path between any two nodes in the PicoDomain graph is slightly over 2 hops. This means the additional visibility that the **GL_GV** detector provides will not contribute significantly more information on the graph structure.

LANL. Here we apply the same 6 detectors to the LANL Comprehensive Cyber Security Events dataset. In a similar manner, we split the data into training and testing sets. The training set consists of 40 days on which no malicious activity is reported, and the testing set of 18 days with malicious activity. This is equivalent to roughly 70% training data, and 30% testing data. Due to the large scale of this dataset, it was necessary that we perform an additional down sampling for the two ML techniques **LOF** and **IF**, which was accomplished by removing timestamps from the training and testing dataset, and removing duplicate events. The TPR and FPR for these two techniques have been adjusted to account for this.

Table 3 shows the results for the six anomaly detectors.

Table 3: Anomaly Detection Results on LANL Dataset

Algorithm	TP	FP	TPR (%)	FPR (%)
UA	542	530082	72	4.4
FL	31	116600	4	1.0
LOF	87	169460	12	9.6
IF	65	299737	9	16.9
GL-LV	503	146285	67	1.2
GL-GV	635	107960	85	0.9

The impact of scale is readily evident in these results, with a significant number of false positives for all detectors, despite reasonably small false-positive rates.

We can see that the **UA** detector performs again reasonably well, with a significant 72% of the malicious authentication events detected. However, with this real-world dataset, we can see how noisy this detector is, with a FPR of 4.4% resulting in over 500,000 false positives. The **FL** detector again fails to perform, indicating that for APT style campaigns, simple failed login attempts are not suitable detectors. Similarly, both ML approaches generated many false positives, and few true positives, again showing that simple authentication events without the added information in the authentication graph are insufficient for malicious authentication detection.

The two graph learning techniques were able to provide the best TPR at the least FPR. The **GL-LV** detector, although returning less true positives than the simple **UA** detector, was still able to detect 67% of the malicious activity, at only 1.2% FPR compared to 4.4% by the **UA** detector. The best performing predictor on this dataset is the **GL_GV** detector, which was able to detect the most malicious authentication events with a TPR of 85%, while maintaining the lowest FPR of 0.9%. For this dataset, the increased context window of the **GL-GV** over the **GL-LV** contributed significantly to the added performance. The average shortest path between any two nodes in the LANL graph is roughly 4 hops. This explains why, in this case, the broader view of the **GL_GV** detector was able to capture more information from the graph structure in the node embeddings, resulting in a better performing link predictor.

It is important to note here that all of the previous experiments were performed on commodity server hardware. Specifically, we utilized a server with two Intel Xeon CPU E5-2683 CPUs, and 512 GB of ram. This provided enough memory and compute power to run any of the detectors discussed on the full 58-day LANL dataset in under 6 hours. We believe that the techniques used here would be supported by the infrastructure already available to our network defenders.

4.4 Reducing False Positives

As we can see from the previous experiment, and specifically Table 3, the effect of false positives on the datasets of the scale found in the real-world can be very detrimental. Even for the best performing detector, the **GL_GV** detector, a false positive rate of 0.9% resulted in over 100,000 individual false positive results in the test data. As these results will ultimately be used by cyber analysts to investigate the threats, it is important that we do our best to keep the false positives to a minimum. In this section, we present some of our observations of the data and results, and design several filters to further reduce the false positive rate by nearly 40%, while reducing true positives by less than 1%.

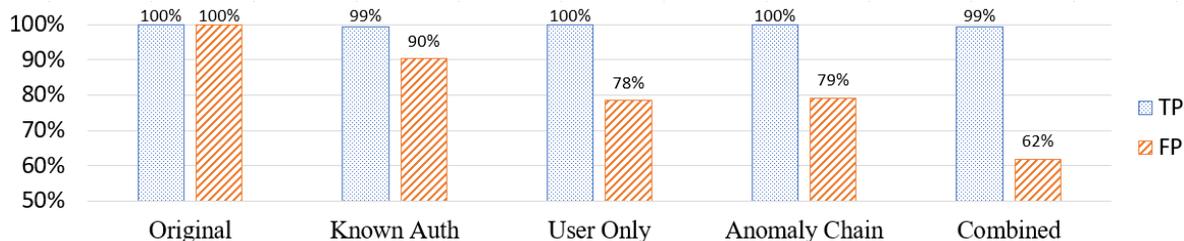


Figure 5: Impact of various approaches in reducing the number of false positives returned on the LANL dataset.

Observation 1: The malicious authentication events are predominantly first authentication events.

This observation was made based on the fact that the simple unknown authentication (UA) detector performed very well at identifying the malicious events. However, its false positive rate was far too high to use on its own. Based on this observation, we use the inverse of this detector as a false positive filter. More precisely, all anomalies generated by the graph learning approach are passed through a filter based on the known authentication events. We discard any of the anomalous authentication events that were previously seen during the training period. This filter corresponds to the "Known Auth" filter in Figure 5. We can see that we achieved about a 10% reduction in false positives, while reducing true positives by less than 1%.

Observation 2: The malicious authentication events are predominantly based on user interactions.

Our authentication graph includes interactions between users and computers, but also interactions between purely computers. Some of the interactions are possibly associated with the red team exercise, however, the labeling scheme utilized by LANL only labeled authentication events involving user accounts as being malicious. Without further details on exactly what the red team activity entailed, it is impossible to label other interactions as malicious or benign that could have been associated with the red team exercise. Based on this, we modify our anomaly detection algorithm, and again add a new filter where the results that are generated and do not involve at least one user account are discarded. This filter corresponds to the "User Only" filter in Figure 5. We can see this had a significant impact on the results, reducing false positives by over 20% from the original, while not reducing the true positives at all.

Observation 3: The malicious authentication events are predominantly related to specific user accounts and systems.

This observation makes sense from a practical standpoint. When an adversary gains access to a network, it is unlikely that they have multiple initial footholds. Typically a single foothold would be established, and then access throughout the network would expand from there. This means that all

of the malicious edges in our authentication graph should be close together, or even form a connected component in the graph. Based on this observation, we build a third filter, where all of the anomalous results are chained together based on their shared nodes and edges. Any anomalous results which do not form a chain with at least one other anomalous event is discarded. This filter corresponds to the "Anomaly Chain" filter in Figure 5. This resulted again in about a 20% reduction in false positives from the original, and no reduction in true positives.

To summarize, the last bars labeled as "Combined" in Figure 5 represent the results when combining all of the previous filters together. We can see this resulted in the best performance, and was able to reduce the number of FPs on the LANL dataset by nearly 40%, while losing < 1% of the true positives.

5 Related Work

This section studies the related works in terms of anomaly detection and node embedding methods.

Anomaly detection for APT identification has been extensively studied. However, the majority of works are based on expensive host-based log analysis, with the goal of anomalous process activity, indicative of malware or exploitation [23] [6] [24] [16]. Some go so far as mining information from user-driven commands for anomaly detection [14]. While host logs may be available in some environments, it would be a significant burden for most large enterprises to capture and store verbose host-based logs such as system call traces.

At the network level, there are techniques for detecting web-based attacks [13], as well as botnet activity [1] utilizing anomaly detection algorithms. A highly related technique [4] combines host information with network information to detect lateral movement. However, they require process-level information from hosts, making this technique a poor fit at the enterprise scale. As lateral movement detection is such a hard problem, some approaches instead focus on detecting the degree to which environments are vulnerable to lateral movement attacks [10].

There are also approaches that look for deviations from known, specification-driven, rules of how an environment

should behave, such as Holmes [20] and Poirot [19]. While these work reasonably well and are able to reduce false positives by explicitly defining what behavior is deemed malicious, they are still based on knowledge derived from a human, and thus risk circumvention by new and novel attack paths. In addition, these techniques require constant maintenance and upkeep to develop new specifications for the constantly evolving attack surface.

Node embedding methods aiming at learning representative embeddings for each node in a graph have been successfully applied to various downstream machine learning tasks, such as node classification [22], link prediction [7], and node recommendation [30]. Existing methods usually take two steps to generate node embeddings. First, they sample meaningful paths to represent structural information in the graph. Second, they apply various data mining techniques from domains such as natural language processing (NLP), utilizing technologies such as word2vec [18] for learning meaningful vector embeddings.

The major difference between existing methods lie in the first step, i.e., how to mine better paths to capture the most important graph information. In this context, the early work DeepWalk [22] applies random walks to build paths for each node. In order to give more importance to close-by neighbors, Line [27] instead applies a breadth-first search strategy, building two types of paths: one-hop neighbors and two-hop neighbors. Further, the authors of node2vec [7] observe that the node embeddings should be decided by two kinds of similarities, homophily and structural equivalence. The homophily strategy would embed the nodes closely that are highly interconnected and in similar cluster or community, while the structural equivalence embeds the nodes closely that share similar structural roles in the graph. Based on these strategies, node2vec implements a biased random walk embedding process which is able to model both similarity measures.

There are additionally many other graph neural network architectures recently proposed, such as the convolution-based GCN [12], attention-based GAT [28], and many variants based on both [9]. However, they are mostly designed for semi-supervised or supervised tasks, and are not as suitable for unsupervised learning as the random-walk based approaches mentioned previously.

6 Limitations & Future Work

Although our results are promising, there are several limiting factors of our approach. The first limitation is the problem of explainability, which is not specific to our technique, but rather a limitation of machine learning techniques in general. When our graph learning algorithms label an event as an anomaly, it is relatively challenging to determine why it has done so. There is current and active research on explaining machine learning and AI algorithms [3], and many even specific to explaining the results of graph learning algorithms in partic-

ular [29]. We may be able to use some of these techniques in the future which would allow us to identify what nodes were most important when generating both the embedding, and ultimately the link prediction scores.

Our detection algorithm is based on the assumption that we will have historic data for each entity we plan to perform link prediction on in the future. If we have never seen an entity authenticate before, then we will not have an embedding generated for that entity, and thus we will be unable to perform the link prediction. There are many ways to handle this problem, such as assigning new entities a generic "new node" embedding, or assigning the new node embedding to the average embedding of its neighbors (provided that they have embeddings themselves), however we have not explored the impact of these various approaches. We believe that, at least in the case of enterprise network authentication, it is a fair assumption to believe that for the vast majority of user accounts in the network, there should be some history of their behavior provided a sufficiently long historic window.

In this work we focused specifically on log data pertaining to authentication events. However, there is a myriad of additional data that we could add to our graph and ultimately to our graph learning algorithms. In the future we plan to add finer grained detail of actions performed by users, such as DNS requests and file-share accesses. This will allow us to also expand our detection algorithm to identify other stages of the kill chain beyond lateral movement, such as command and control traffic, which would likely cause anomalous DNS requests.

7 Conclusion

In this work we discussed the challenging problem of detecting lateral movement of APT-level adversaries within enterprise computer networks. We explained why existing signature-based intrusion detection techniques are insufficient, and existing behavioral analytics are too fine grained. We introduced our technique of abstracting a computer network to a graph of authenticating entities, and performing unsupervised graph learning to generate node behavior embeddings. We discussed how we use these embeddings to perform link prediction, and ultimately anomaly detection for malicious authentication events. We applied our techniques to both simulated and real-world datasets and were able to detect anomalous authentication links with both increased true positive rates, and decreased false positive rates, over rule-based heuristics and non-graph ML anomaly detectors. We analyzed the results of our algorithm, and developed several simple filters to further reduce the false positive rate of our technique.

Acknowledgment

This work was supported in part by DARPA under agreement number N66001-18-C-4033 and National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774, as well as support from the ARCS Foundation. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense, National Science Foundation, ARCS, or the U.S. Government.

References

- [1] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.
- [2] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.
- [3] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. Towards explanation of dnn-based prediction with guided feature inversion. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1358–1367, 2018.
- [4] A. Fawaz, A. Bohara, C. Cheh, and W. H. Sanders. Lateral movement detection using distributed data fusion. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30, Sep. 2016.
- [5] FireEye. M-trends 2019. <https://content.fireeye.com/m-trends/rpt-m-trends-2019>, 2019.
- [6] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.
- [7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.
- [8] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.
- [9] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.
- [10] John R Johnson and Emilie A Hogan. A graph analytic metric for mitigating advanced persistent threat. In *2013 IEEE International Conference on Intelligence and Security Informatics*, pages 129–133. IEEE, 2013.
- [11] Alexander D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.
- [12] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [13] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261. ACM, 2003.
- [14] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pages 120–132. IEEE, 1999.
- [15] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.
- [16] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1035–1044. ACM, 2016.
- [17] Lockheed Martin. The cyber kill chain. <https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html>, 2020. Accessed: 2020-01-16.
- [18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
- [19] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, page 1795–1812, New York, NY, USA, 2019. Association for Computing Machinery.

- [20] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.
- [21] MITRE. Mitre att@ck. <https://attack.mitre.org/>, 2020. Accessed: 2020-01-16.
- [22] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.
- [23] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 144–155. IEEE, 2000.
- [24] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.
- [25] Snort. Snort. <https://www.snort.org/>, 2020. Accessed:2020-01-16.
- [26] Splunk. Splunk. <https://www.splunk.com>, 2020. Accessed:2020-01-16.
- [27] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.
- [28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
- [29] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnn explainer: A tool for post-hoc explanation of graph neural networks. *arXiv preprint arXiv:1903.03894*, 2019.
- [30] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. Personalized entity recommendation: A heterogeneous information network approach. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 283–292, 2014.
- [31] Zeek. The zeek network security monitor. <https://zeek.org>, 2020. Accessed: 2020-01-16.

An Object Detection based Solver for Google’s Image reCAPTCHA v2

Md Imran Hossen* Yazhou Tu* Md Fazle Rabby* Md Nazmul Islam* Hui Cao† Xiali Hei*

*University of Louisiana at Lafayette

†Xi’an Jiaotong University

Abstract

Previous work showed that reCAPTCHA v2’s image challenges could be solved by automated programs armed with Deep Neural Network (DNN) image classifiers and vision APIs provided by off-the-shelf image recognition services. In response to emerging threats, Google has made significant updates to its image reCAPTCHA v2 challenges that can render the prior approaches ineffective to a great extent. In this paper, we investigate the robustness of the latest version of reCAPTCHA v2 against advanced object detection based solvers. We propose a fully automated object detection based system that breaks the most advanced challenges of reCAPTCHA v2 with an online success rate of 83.25%, the highest success rate to date, and it takes only 19.93 seconds (including network delays) on average to crack a challenge. We also study the updated security features of reCAPTCHA v2, such as anti-recognition mechanisms, improved anti-bot detection techniques, and adjustable security preferences. Our extensive experiments show that while these security features can provide some resistance against automated attacks, adversaries can still bypass most of them. Our experiment findings indicate that the recent advances in object detection technologies pose a severe threat to the security of image captcha designs relying on simple object detection as their underlying AI problem.

1 Introduction

CAPTCHA is a defense mechanism against malicious bot programs on the Internet by presenting users a test that most humans can pass, but current computer programs cannot [49]. Often, CAPTCHA makes use of a hard and unsolved AI problem. Over the last two decades, text CAPTCHAs have become increasingly vulnerable to automated attacks as the underlying AI problems have become solvable by computer programs [17, 19, 20, 27, 28, 34, 35, 51, 51–53]. As a result, text CAPTCHAs are no longer considered secure. In fact, in March 2018, Google shut down its popular text CAPTCHA scheme reCAPTCHA v1 [23]. Image CAPTCHA schemes

have emerged as a superior alternative to text ones as they are considered more robust to automated attacks.

reCAPTCHA v2, a dominant image CAPTCHA service released by Google in 2014, asks users to perform an image recognition task to verify that they are humans and not bots. However, in recent years, deep learning (DL) algorithms have achieved impressive successes in several complex image recognition tasks, often matching or even outperforming the cognitive ability of humans [30]. Consequently, successful attacks against reCAPTCHA v2 that leverage Deep Neural Network (DNN) image classifier and off-the-shelf (OTS) image recognition services have been proposed [44, 50].

The prior work advanced our understanding of the security issues of image CAPTCHAs and led to better CAPTCHA designs. However, recently, Google has made several major security updates to reCAPTCHA v2 image challenges that can render prior image classification and recognition based approaches ineffective to a great extent. For example, the latest version of reCAPTCHA pulls challenge images from relatively complex and common scenes as opposed to monotonic and simple images in the past. Through a comprehensive experiment, we show that both image classifiers and image recognition APIs provide poor success rates against the latest reCAPTCHA v2 challenges.

Our experiment also shows that the current version of reCAPTCHA v2 adopts several additional security enhancements over the earlier versions. First, reCAPTCHA v2 has introduced anti-recognition techniques to render the challenge images unrecognizable to state-of-the-art image recognition technologies. For example, it often presents noisy, blurry, and distorted images. reCAPTCHA image challenges are likely to be using adversarial examples [15, 46] as a part of the anti-recognition mechanism as well. Second, it adapts the difficulty-level for suspicious clients by presenting them with harder challenges. Third, the improved anti-bot detection mechanism of reCAPTCHA can now detect the popular web automation framework like Selenium. Apart from those, reCAPTCHA v2 also added click-based CAPTCHA tests, which are not explored in the prior studies. We suspect that

the click-based CAPTCHAs were not available at the time of publication of the most recent attack on reCAPTCHA v2.

Taking reCAPTCHA v2 as an example, we investigate the security of image CAPTCHA schemes against advanced object detection technologies. To this end, we develop an object detection based real-time solver that can identify and localize target objects in reCAPTCHA’s most complex images with high accuracy and efficiency. Specifically, our system can break reCAPTCHA image challenges with a success rate of 83.25%, the highest success rate to date, and it takes only 19.93 seconds (including network delays) on average to crack a challenge. Our economic analysis of human-based CAPTCHA solving services shows that our automated CAPTCHA solver provides comparable performance to human labor. Therefore, the scammers can exploit our system as an alternative to human labor to launch a large-scale attack against reCAPTCHA v2 for monetary or malicious purposes, leaving millions of websites at the risk of being abused by bots [11].

We also provide an extensive analysis of the security features of the latest version of reCAPTCHA v2. First, we find that the anti-recognition mechanisms employed by reCAPTCHA can significantly degrade the performance of both image recognition and object detection based solvers. However, our extensive analysis shows that we can neutralize reCAPTCHA’s anti-recognition attempts by applying advanced training methods to develop a highly effective object detection based solver. Second, we also find that our system can bypass many other imposed security restrictions. For example, we can bypass the browser automation framework restriction by using the puppeteer-firefox [10] framework. Our findings reveal that despite all the evident initiatives by Google, reCAPTCHA still fails to meet the stringent security requirements of a secure and robust CAPTCHA scheme.

In summary, we make the following contributions:

- Through extensive analysis, we show that prior DNN image classifiers and off-the-shelf vision APIs based approaches are no longer effective against the latest version of reCAPTCHA v2. We then propose an object detection based attack that can break the most advanced image challenges provided by reCAPTCHA v2 with high accuracy and efficiency.
- We provide a comprehensive security analysis of different security features employed by the latest version of reCAPTCHA v2. Our extensive study shows that these features can provide some resistance to automated attacks. However, adversaries can still bypass most of them.
- Our study indicates that the recent advances in object detection algorithms can severely undermine the security of image CAPTCHA designs. As such, the broader impact of our attack is that any image CAPTCHA schemes relying on simple object detection as their underlying AI problem to make a distinction between bots and humans might be susceptible to this kind of attack.

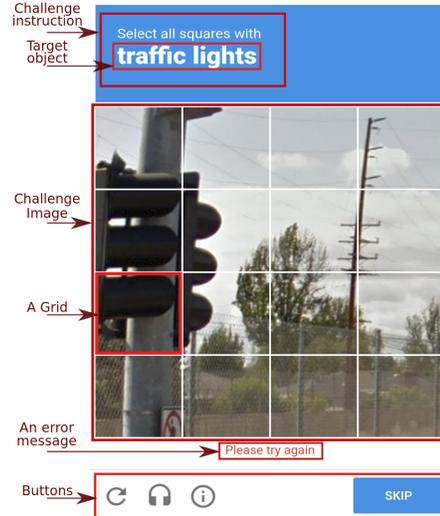


Figure 1: A reCAPTCHA v2 challenge widget.

2 reCAPTCHA v2 background

reCAPTCHA v2 relies on an *advanced risk analysis engine* to score users’ requests and let legitimate users bypass the CAPTCHA test. Once the user clicks the reCAPTCHA – “I’m not a robot” — checkbox, the advanced risk analysis engine tries to determine whether the user is a human using various signals collected by the system, including different aspects of the user’s browser environment, and Google tracking cookies [36, 44]. If the system finds the user suspicious, it asks the user to solve one or more image CAPTCHA(s) to prove that he/she is a human and not a bot. In general, a user with no history with Google services will be assigned to relatively difficult challenges. In this paper, our system attempts to solve these CAPTCHAs. Note that, Bock *et al.* followed a similar approach to break reCAPTCHA’s audio challenges in 2017 [16].

It is important to note that the third version of reCAPTCHA, reCAPTCHA v3, was released in October 2018. reCAPTCHA v3 is intended to be frictionless, *i.e.*, not requiring any users’ involvement in passing a challenge. However, it has raised some serious security concerns due to the method it uses to collect users’ information [21, 42]. In this paper, we only target reCAPTCHA v2’s most recent (as of March 2020) image challenges because it is still the most popular and widely used version of reCAPTCHA deployed on the Internet. From now on, we will use the term reCAPTCHA to refer to reCAPTCHA v2 unless otherwise specified.

Challenge widget. If reCAPTCHA requires the user to solve a challenge, a new `iframe` gets loaded on the webpage after clicking on the “I’m not a robot” checkbox. The `iframe` contains the actual reCAPTCHA challenge (Figure 1). The challenge widget can be divided into three sections: top, middle, and bottom. The top section includes instructions about

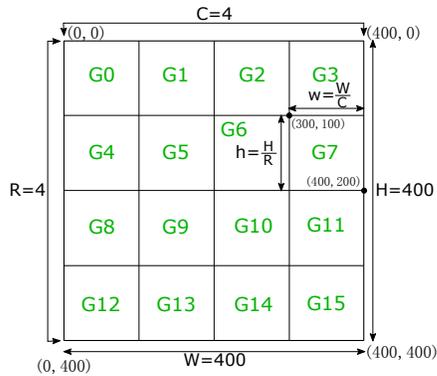


Figure 2: Representing a 400px \times 400px challenge image as an $R \times C$ grid. Here, R =No. of rows, and C =No. of cells per row in the HTML table element holding the challenge image.

how to solve the challenge. The section in the middle holds the candidate images. The user has to select or click on images that contain the target object mentioned in the instruction. At the bottom, it has multiple buttons, including the “reload” button, “audio CAPTCHA” button, and the “verify” button.

The images are located inside an HTML table element. The table has multiple rows, and each row holds the same number of cells. Each cell has an `img` tag in it and renders an image from the URL specified in the tag. If the table has 4 rows and each row has 4 cells, then 16 candidate images in total will be rendered, from which the user has to select the right images. However, all these images are pulled from a single source URL in the challenge widget initially, *i.e.*, a single image is split across multiple table cells in an equal proportion. Therefore, this particular challenge image can be treated as a 4×4 grid. Figure 2 illustrates the process of representing a 400px \times 400px challenge image as an $R \times C$ grid. Here R and C correspond to the number of rows and the number of cells per row in the table element.

reCAPTCHA CAPTCHA types. The current version of reCAPTCHA has two types of image CAPTCHAs: 1) *selection-based image CAPTCHA* and 2) *click-based image CAPTCHA*. The selection-based CAPTCHA requires the user to select the correct grids containing the right object as specified in the instruction to pass the challenge (see Figure 6 in Appendix A). It is the common CAPTCHA type that the user would encounter in a reCAPTCHA-protected site. The click-based CAPTCHAs have been introduced only recently. In a click-based image challenge, when the user clicks on a grid, the image on the grid disappears, and a new image gets generated in its place (see Figure 7 in Appendix A). The user has to repetitively click on the potential grids until the target object is no longer present in any of the grid while submitting a challenge. It takes a relatively long time to solve the click-based CAPTCHAs than selection-based ones as there is a delay between the click and image regeneration process.

3 Threat model

We assume the attacker’s goal is to abuse Web applications protected by reCAPTCHA using an automated program. We also assume the attacker has access to a GPU enabled machine to deploy an object detection system for cracking CAPTCHAs. The attacker can launch the attack from a single IP address. However, having access to a large IP pool will allow the attacker to launch a large-scale attack. reCAPTCHA may occasionally block an IP address for some time. In such a scenario, the attacker may need to use a proxy service or anonymity network such as Tor [24] to bypass the IP restriction. In summary, we consider a low-to-moderately resourced attacker whose goal is to deploy a highly effective automated solver to break reCAPTCHA challenges for malicious purposes.

4 Our approach

Our automated CAPTCHA breaker consists of a browser automation module and a solver module.

Browser automation module. The *browser automation module* is responsible for automating different browser-specific tasks while solving a CAPTCHA challenge. These tasks include locating the reCAPTCHA checkbox, initiating the reCAPTCHA challenge, and identifying the potential HTML elements on the challenge widget. This module is also in charge of fetching challenge images, submitting the solution once the CAPTCHA is solved, monitoring the progress of the challenge, and checking the reCAPTCHA verification status.

Solver module. The solver module consists of two main components: the base object detector and the *bounding box to grid mapping* algorithm. The base object detector takes the challenge image from the *browser automation module* and identifies and localizes objects in it. For each recognized object instance, the base detector returns its class name, the confidence score, and coordinate information in terms of the bounding box. The *bounding box to grid mapping* algorithm then uses this data to map the bounding boxes holding the target object back to the grids where they are present.

4.1 The browser automation module

The *browser automation module* first visits a reCAPTCHA-protected webpage and locates the frame element holding “I am not a robot” checkbox. It then clicks on the checkbox, which is identified by `recaptcha-anchor`, to initiate the challenge. Now our system switches to the challenge widget. Then it primarily conducts the following steps to solve the challenge.

Extracting challenge instruction. Our system locates the element holding the challenge instruction `rc-imageselect-instructions`. The challenge instruction is a multi-line string, and the second line always refers to the name of the target object. Further, it indicates the challenge type. For instance, in click-based image

CAPTCHAs, the challenge instruction always holds the phrase — “Click verify once there are none left” (See Figure 7). The name of the target object, for which we must solve the challenge, may not always be in the singular form. If that happens, we singularize it.

Determining the total number of rows (R) and the number of cells per rows (C). As discussed in Section 2, if we know the total number of rows (R) and the number of cells per row (C) in the HTML table holding the challenge image, we can represent the challenge image as an $R \times C$ grid. We use JavaScript methods to determine R and C .

Downloading challenge image. The element `rc-imageselect-tile` holds the challenge image. There will be multiple such elements based on the total number of grids. Since all the elements link to the same image, our system downloads the first image only. However, for click-based CAPTCHAs, it will need to download dynamically loaded images on the selected grids as well.

Identifying buttons on the challenge widget. To submit the challenge once it is solved, we need to click on the “verify” button. Our system locates the “verify” button using its identifiers `recaptcha-verify-button`.

4.2 Implementation of the solver module

The solver module identifies and localizes target objects in reCAPTCHA challenge images. Further, the module is responsible for mapping the detected objects back to their corresponding (potential) grids in the original challenge. The two main components of this module are a *base object detection system* and the *bounding box to the grid mapping* algorithm. We now discuss each of them in detail.

4.2.1 Base object detector: YOLOv3

We use YOLOv3 as the base object detector after experimenting with several other advanced object detectors, including Faster R-CNN [40], R-FCN [22], SSD [33], and RetinaNet [32]. We find YOLOv3 to be significantly faster than all other tested object detectors when running the detection on a test image; however, the accuracy of YOLOv3 might be slightly lower than other object detectors. Since solving CAPTCHAs is a time-sensitive task, we opt to use YOLOv3 for its superior speed. The feature extractor network in YOLOv3 is called Darknet-53 because it has 53 convolutional layers, with shortcut connections. See [39] for details.

Datasets. We use two datasets, specifically developed to handle object categories found in reCAPTCHA challenges. The first dataset is a publicly available dataset called MS COCO [13]. The MS COCO dataset has 80,000 training images and 40,000 validation images with 80 object classes, out of which 8 classes frequently appear in reCAPTCHA challenges. The MS COCO object classes common to re-

CAPTCHA object categories are bicycle, boat, bus, car, fire hydrant, motorcycle, parking meter, and traffic light. The second dataset is a custom one that we develop by ourselves. We crawled over 6,000 images from different sources such as Flickr¹, Google image search², and Bing image search³. After preprocessing these, we end up with 4800 images. We also use 2100 images from the original reCAPTCHA challenges for this dataset. We manually annotated and labeled the object instances in those images to prepare and finalize the dataset. Our final custom dataset has 11 object categories: boat, bridge, chimney, crosswalk, mountain, palm tree, stair, statue, taxi, tractor, and tree.

Training the base object detector. We use two YOLOv3 models trained on the two datasets. We mostly go with the default architecture of the YOLOv3 network with some minor modifications for both models. We set up the batch size to 64, and the learning rate to 0.001 for training. We use the Darknet [38], an open-source neural network framework written in C, for training the YOLOv3 models. We train the model on the MS COCO dataset for roughly 15 days, and the model on the custom dataset for 2 days. The training is performed on a server with an NVIDIA RTX 2070 GPU. We then evaluate the weight files for both models on corresponding test sets and choose the best weights. Our final model for the MS COCO dataset has the mean average precision at 0.5 IOU (mAP@.5) of 57.4% on the testing set. The second model has obtained a mAP@.5 value of 51.79% on the respective testing set.

Inference or making predictions. We use the Darknet framework to make predictions on reCAPTCHA challenge images with our trained models. By default, Darknet does not provide any localization information. We adjust the source code to output the bounding box coordinates when running the inference on an image. The modified prediction output includes class name, confidence score, and bounding box coordinates for each detected object instance in a prediction operation. We set the detection threshold to 0.2.

4.2.2 The bounding box to grid mapping algorithm

After detecting the objects with the base object detector in the challenge image, we need to map the objects back to their corresponding grids in the original challenge. Our *bounding box to grid mapping* algorithm works as follows.

1. Use the R and C parameters from the *browser automation module* to get an $R \times C$ grid representation of the image.
2. Compute coordinates of each grid relative to the top left of the image (see Figure 2).
3. Take the prediction output from the base object detector.

¹<https://www.flickr.com/>

²<https://images.google.com/>

³<https://www.bing.com/images/>

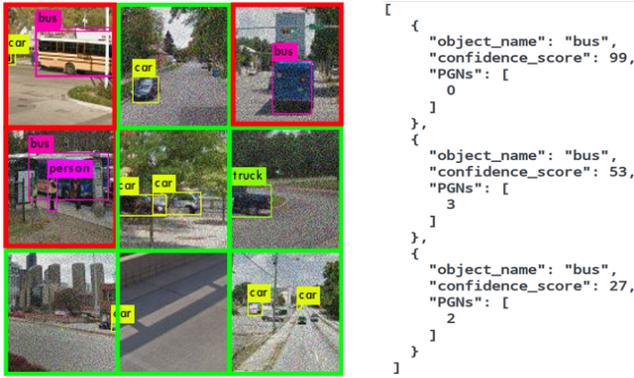


Figure 3: A result returned by the *bounding box to grid mapping* algorithm. Left: An original challenge image (the target object is a “bus”). Right: The JSON array returned by the algorithm.

4. For each bounding box with the class label matching the target object name in the challenge, take the coordinates of the box and the grids. If any of the coordinates of the bounding box falls inside a grid, mark it as a potential grid. Depending on the size of the bounding box, it may fall within multiple grids. We store all of these grid numbers in an array and call it as the potential grid numbers (PGNs).
5. For each bounding box, return the class name, confidence score, and the PGNs.
6. Return the results as a JSON array.

Figure 3 shows an example of the result returned by the *bounding box to the grid mapping* algorithm for a sample challenge image.

4.3 Submitting and verifying challenges

The JSON array returned by the solver module is passed to the *browser automation module*. The browser automation module first extracts the potential grid numbers from the PGNs arrays and locates the representative grids in the HTML table. It then clicks on these grids in the challenge widget and finally clicks the “verify” button when the process is completed.

The system then verifies whether the challenge is passed or not using the “reCAPTCHA ARIA status messages.”⁴ We further verify the challenges submitted to our own websites by validating user response token, `g-recaptcha-response`, to the reCAPTCHA backend. The `g-recaptcha-response` remains empty until the challenge is solved. When a challenge is successfully solved, it gets populated with a long string. After submitting a challenge to our website, our bot first extracts the user response token. It then sends a verification request

⁴https://support.google.com/recaptcha/#aria_status_message

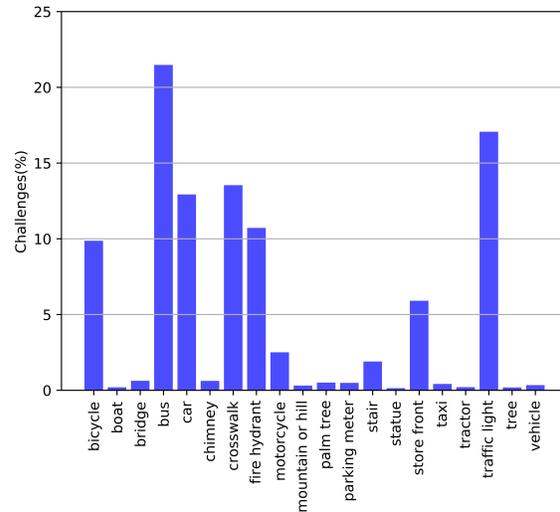


Figure 4: The frequency of different object categories in collected challenges.

to the reCAPTCHA backend server with this token and the secret key to authenticate the token.

5 Attack evaluation

5.1 Implementation and evaluation platform

The *browser automation module* is built upon the puppeteer-firefox [10], a node library developed by Google, to control the Firefox web browser programmatically. The core functionalities of the module are developed using JavaScript. The base object detector in the solver module is based on the YOLOv3 object detection algorithm. We train and test the YOLOv3 models with a customized version of the Darknet framework that especially meets our needs. Our *bounding box to the grid mapping* algorithm is written in C for efficiency.

We train the YOLOv3 models on a server with 6 Intel® Xeon® E5-2667 CPUs, an NVIDIA GeForce RTX 2070 GPU, and 96GB of RAM running the Arch Linux operating system. We compile the Darknet framework against the GPU with CUDA 10.2 and cuDNN 7.5. We conduct all the experiments on this machine.

Experimental setting. We use puppeteer-firefox (version 0.5.1) running on top of node library version 14.4.0 for browser automation. The browser we used is Firefox 65.0. We run the web browser in a clean state each time, *i.e.*, no caches or cookies are retained between two subsequent requests. We do not attempt to obfuscate any aspects of the requests or web browser properties (e.g., using custom User-Agent header or modifying the Navigator object properties). Further, unless otherwise specified, all the requests to reCAPTCHA-protected websites are made from a single IP address and a single machine.

Table 1: Performance of image recognition services.

Image Recognition Service	Success Rate (%)	Speed (s)
Google Cloud Vision	19	16.67
Microsoft Azure Computer Vision	34	17.90
Amazon Rekognition	47	19.68
Clarifai	27	15.35

Table 2: Performance of object detection services.

Object Detection Service	Success Rate (%)	Speed (s)
Google Cloud Vision	36	9.47
Microsoft Azure Computer Vision	31	10.54
Amazon Rekognition	38	9.86

5.2 Preliminary analysis

To do the preliminary analysis, we collect 10,385 reCAPTCHA challenges from 8 websites protected by reCAPTCHA service from May 2019 to November 2019. Figure 4 shows frequencies of different object categories in the collected challenges. As we can see, there are only 19 object categories, with the top 5 categories representing over 75% of the total challenges. The top 5 object classes are bus, traffic light, crosswalk, car, and fire hydrant.

Breaking reCAPTCHA with vision APIs for image recognition. We test 4 popular off-the-shelf online vision APIs for image recognition. The services we use are Cloud Vision API provided by Google [7], Azure Computer Vision API provided by Microsoft [9], Rekognition API provided by Amazon [2], and the API provided by Clarifai [5]. First, we select some challenge images from different categories, extract the individual grids from them, and submit those grids to the image recognition services to analyze the tags (labels) returned by them. We find that in most cases, one of the labels for a grid holding the target object matches precisely with the name of the object in the reCAPTCHA challenge instruction, thus simplifying the process of mapping the tags returned by an API service to reCAPTCHA challenge object names while submitting a challenge. However, we find one instance where the labels are not consistent across various APIs. For example, Amazon Rekognition API classifies reCAPTCHA’s crosswalk images as “Zebra Crossings,” while Google’s Cloud Vision API recognizes them as “Pedestrian crossings.” We do a simple preprocessing that transforms these labels to name “crosswalk” for consistency.

Next, we develop a proof-of-concept attack and submit 100 live reCAPTCHA challenges separately using each image recognition API. Table 1 provides the success rate and speed of attack using image recognition services. The Google Cloud Vision provides the lowest success rate, followed by Clarifai. We note that the attack success rate is below 40% for all the services except for Amazon Rekognition.

Finally, we manually verify the results and analyze the failed challenges. We find that the image recognition services’ poor performance is due to the complex nature of the current challenge images, which often contain complex everyday scenes with common objects in their natural context. For example, we find many instances where a potential grid holding a “crosswalk” also holds other common objects such as “car” and “traffic light” in it, and tags returned by an API include names of all the objects except the primary target. Further, in many challenges, a single target object spans across multiple grids, and some of those grids contain only a tiny part of the whole object. Image recognition services failed to identify the target object in such a scenario. The earlier version of reCAPTCHA used to show relatively simple images, usually containing one disparate object per grid or images with simple scenes having a monotonic background, making it easier for image recognition services to analyze the contents.

Breaking reCAPTCHA with an image classifier. We also perform an attack using a Convolutional Neural Network (CNN) based image classifier. The classifier is trained on over 98,000 images from 18 classes. These include all object classes in Figure 4, except the “store front.” Interestingly, the “store front” class has been phased-out from reCAPTCHA challenges during the later part of our data collection period. We then submit 100 live reCAPTCHA challenges using our image classifier based solver. The success rate and speed of attack are 21% and 16.96 seconds, respectively. After analyzing the failed challenges manually, we find that the same factors related to the poor performance of the image recognition APIs contributed equally (or even higher) to the low success rate of the image classifier based attack.

Breaking reCAPTCHA with online vision APIs for object detection. We also carry out a proof-of-concept attack using three off-the-shelf computer vision APIs for object detection provided by Google, Microsoft, and Amazon. We customize our *bounding box to the grid mapping* algorithm to process the bounding box results for the objects detected by the APIs. Like before, we submit 100 live reCAPTCHA challenges using these APIs. Table 2 shows attack performance of each off-the-shelf object detection API. We can see that the Amazon Rekognition API and Google Cloud Vision API achieve similar performance, while Microsoft Azure Computer Vision API performs relatively poorly. We analyze the results to understand why these services are not effective against reCAPTCHA challenges. We find several factors that contribute to low success rates. First, these services can recognize objects from certain object categories only. However, most of them can detect objects that frequently appear in the reCAPTCHA challenges such as “bus”, “car”, “traffic light”, and “bicycle”. While the top 5 objects in Figure 4 account for around 70% of the submitted challenges during this experiment, our manual analysis shows that the object detection APIs fail to identify at least one target object in these categories in most of the

Table 3: The time required to break a reCAPTCHA challenge by percentiles.

Percentile	1st	5th	95th	99th
Speed (s)	13.28	13.91	39.76	58.65

failed cases. It suggests that the cloud-based vision APIs for object detection are still in their early stage of development, and yet to be ready to handle complex images such as those found in reCAPTCHA challenges.

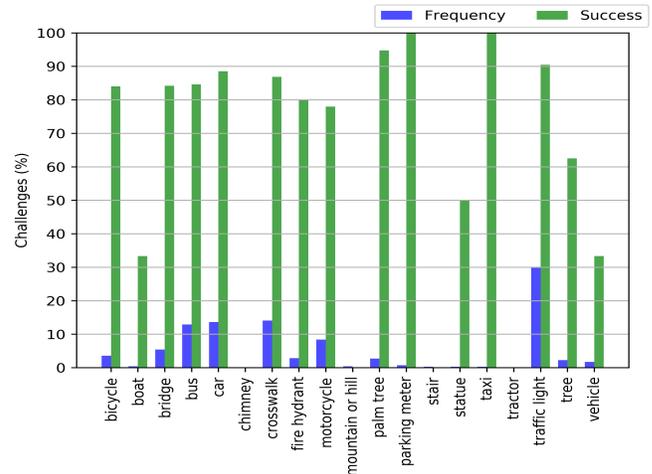
5.3 Breaking reCAPTCHA challenges with our system

Success rate and speed of attack. To evaluate the effectiveness and efficiency of our approach, we submit 800 challenges to 4 reCAPTCHA-enabled websites using our automated CAPTCHA-breaking system. Out of them, 701 challenges are selection-based, 87 challenges are click-based, and 12 are “no CAPTCHA reCAPTCHA” challenges, where our system gets verified simply by clicking on the reCAPTCHA checkbox. Our system breaks 656 (out of 788) challenges, resulting in a success rate of 83.25%.

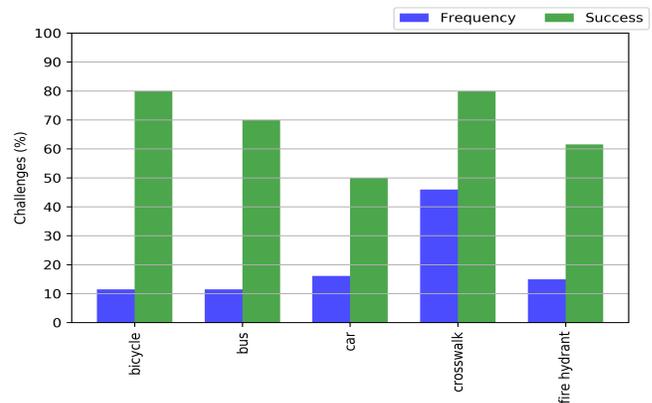
The average speed of breaking a CAPTCHA challenge is 19.93 seconds, including delays. The delays include network delay to load and download images, which takes about 1–8 seconds depending on CAPTCHA types and artificially induced delay between each of the clicks. The minimum, median, and maximum time needed to break a challenge are 13.11, 14.92, and 89.02 seconds respectively. Table 3 lists the time required to break a challenge by percentiles. Our solver module takes about 6.5 seconds to detect objects in a challenge image regardless of the number of objects being present.

Attack on selection-based CAPTCHAs. Generally, the selection-based image reCAPTCHA challenges appear more often than click-based ones. The success rate and speed of breaking selection-based challenges are 84.74% and 17.47 seconds, respectively. Figure 5a shows frequency and success rate for each object category in the submitted challenges. As we can see, only 19 object categories have been repeated across all 701 selection-based reCAPTCHA image challenges. Further, the top 5 object categories constitute over 78% of the total challenges. If we consider the first 10 categories, this number goes above 95%.

Note that reCAPTCHA often asks users to solve multiple image puzzles in a row to pass a single selection-based reCAPTCHA test. However, in 80.81% of passed CAPTCHAs, our system is required to solve only one image test. In 16.84% of challenges, it is required to solve 2 image puzzles. The maximum number of puzzles required to pass a test is 5, and it occurs only twice.



(a) Selection-based CAPTCHA.



(b) Click-based CAPTCHA.

Figure 5: Attack performance. Frequency and success rate for each object category.

We find that in the majority of cases, there are at least 3 potential grids required to be chosen to pass a selection-based CAPTCHA test. Precisely, in 5.72% of passed CAPTCHAs, our system is asked to select 2 grids. In 42.59% of solved CAPTCHAs, it is required to choose 3 grids. In 32.15% of solved challenges, the system is required to select 4 grids. The number of selected potential grids in the remaining challenges ranges from 5 to 14. We also find 2 tests where our system had to choose 18 grids to pass the challenges. It takes 4.01 seconds to select a grid while solving a challenge, on average.

Attack on click-based CAPTCHAs. We come across only 87 click-based CAPTCHAs in the 800 submitted challenges, and our system passes 62 of them. The success rate and speed are 71.26% and 43.53 seconds, respectively. Figure 5b provides the frequency and success rate for different object categories in click-based reCAPTCHA challenges. As we can see, in click-based CAPTCHA challenges, there are only five object categories.

Since reCAPTCHA’s click-based CAPTCHAs are relatively new and have not been explored in the previous study, we experiment to analyze the security of it further. The study by Sivakorn *et al.* revealed that the initial implementation of reCAPTCHA v2 used to provide flexibility in its selection-based challenges, *i.e.*, it used to accept 1 or 2 wrong grid selection(s) along with a certain number of correct grid selections while solving a CAPTCHA test [44]. We investigate whether reCAPTCHA provides such flexibility for click-based challenges. We submit some CAPTCHAs by clicking on a different combination of correct and wrong grids. We submit 50 challenges for each combination. The result of our experiment is summarized in Table 4. The highest success rate we achieve is 6% when we click 6 correct grids and 1 wrong grid while submitting the challenges. It suggests that reCAPTCHA does not usually provide any solution flexibility for click-based CAPTCHAs.

Table 4: The success rates of different combinations of correct and wrong grid selection in a click-based CAPTCHA solution.

# of Correct Grid	# of Wrong Grid	Success Rate
3	1	0.00%
4	1	2.00%
5	1	0.00%
6	1	6.00%
6	2	0.00%
3 (out of 4)	0	0.00%
4 (out of 5)	0	0.00%
5 (out of 6)	0	2.00%

Impact of security preference. reCAPTCHA allows the website owners to adjust the security level based on their needs while deploying it to their websites. There are three levels in the security preference setting from “Easiest for user” to “Most secure.” By default, reCAPTCHA uses the security level in the middle, which we call “Medium secure.” To investigate the impact of these settings on the attackers’ success rate, we deploy reCAPTCHA on our website and submit 50 challenges for each security setting. We run the experiment from a network that is isolated from the network hosting our webserver to avoid any biases. We further follow the same experimental setting mentioned in 5.1.

The results of our findings are summarized in Table 5. The difference in the accuracy of our system across three security preferences is negligible. We have not noticed any obvious pattern that can distinguish one security preference from others. However, for the “Easiest for user” setting, we find that reCAPTCHA occasionally accepts a solution even when our bot misses one potential grid containing the target object or clicks on a wrong grid along with the correct grid selections. Further, reCAPTCHA often requires the bot to solve multiple image puzzles in a single selection-based challenge when us-

Table 5: The success rates of different security preferences in the reCAPTCHA deployment setting.

Security Preference	Success Rate (%)	Speed (s)
Easiest for user	82	16.75
Medium secure	78	14.31
Most secure	84	18.79

ing the “Most secure” security preference on the reCAPTCHA admin panel for our website.

Impact of browser automation software. To study the impact of different browser automation frameworks on the performance of the bots, we develop a bot using the Selenium [12]. Selenium is one of the most widely used browser automation frameworks, which was also used by prior arts. Selenium provides WebDriver for both Mozilla Firefox and Google Chrome web browsers. In particular, we use Selenium Python bindings (version 3.141.0) with Python version 2.7.18 in this experiment. For web browsers, we select Firefox 65.0 and Chrome 78.0.3882.0. To keep the experiment consistent with our main attack, we run the program from the same machine, and we access reCAPTCHA-enabled websites from the same IP address. Further, we clear the caches and cookies each time we launch the program.

First, we use Firefox WebDriver. We submit 100 CAPTCHAs and notice that most of our attempts fail to break them. Accurately, the system can solve only 32% of the total submitted challenges. A careful examination of our system log reveals that reCAPTCHA rejects many of the potentially correct solutions. Further, 12 out of 100 requests have been blocked with the message — “We’re sorry, but your computer or network may be sending automated queries. To protect our users, we can’t process your request right now.” Note that at the same time, we also run our original puppeteer-firefox based system and verify that it can normally solve the challenges. Next, we repeat the same experiment with Chrome WebDriver. We recognize a similar pattern as before: the success rate of breaking the CAPTCHAs is below 40%. We also find that reCAPTCHA shows a significantly higher percentage of click-based CAPTCHAs when we use the Selenium. Specifically, more than 25% of the challenges that our system attempt to solve are click-based ones. Furthermore, we also encounter many noisy images.

Since reCAPTCHA’s advanced risk analysis engine treats our Selenium based system as highly suspicious, we try to obfuscate the presence of Selenium and investigate whether the obfuscation could improve our attack performance. When using an automation framework, the browser is supposed to set `navigator.webdriver` property to “true” according to W3C specification. However, an adversary may not follow this specification in an attempt to hide the presence of WebDriver to dodge detection. To experiment with an attacker’s perspective, we set this property to “false.” Moreover, we ob-

fuscate different aspects of the browser environment, such as the user-agent string, the number of plugins used, the number of fonts, and screen resolution. We then run a series of experiments with these settings. However, none of our attempts have resulted in any noticeable difference.

Impact of anti-recognition. We conduct a comprehensive analysis of the anti-recognition techniques employed by reCAPTCHA that have been introduced only recently in an attempt to undermine AI-based recognition. First, we need to identify noisy, distorted, or perturbed images. However, identifying such images is not trivial. A simple approach could be using Signal to Noise Ratio (SNR) measure to estimate the noise in an image. However, we cannot simply use SNR because we do not have access to the original ground truth images. In this situation, we find scikit-image’s [48] `estimate_sigma` function to be particularly useful. `estimate_sigma` provides a rough estimation of the average noise standard deviation (σ_{noise}) across color channels for an RGB image. We calculate σ_{noise} for all 1,048 images that our solver attempt to solve during our experiment. The σ_{noise} for 928 (over 88%) images are all below 2. For the remaining 120 images, the σ_{noise} ranges from 2 to 14.86. By manually analyzing the images, and doing the visual inspection, we find images with σ_{noise} over 10 appear to be extremely noisy and distorted, and contains some kinds of perturbations. We find 52 such images, and we use them for our comprehensive analysis of reCAPTCHA’s anti-recognition attempts.

We suspect that reCAPTCHA might be using adversarial examples [15, 46], slightly perturbed images maliciously crafted to fool pre-trained image classifiers. Prior work also recommended using adversarial examples to limit the impact DNN image classifier based attacks [44]. We experiment to validate our assumption. First, we labeled the potential grids in images with perturbations. We then extract these grids from the images and submit them to the computer vision services for recognition.

Table 6 shows the label sets returned by 4 off-the-shelf image recognition services for an actual reCAPTCHA challenge image. The target object in the challenge is a fire hydrant, and the potential grids are 1, 3, and 4. Table 7 shows the result of our experiment. We can see that vision APIs have misclassified a significant number of potential grids. Clarifai API has the highest number of misclassifications, followed by Microsoft Computer Vision API. Google Cloud Vision API did not return any labels for 32 potential grids. Note that we consider a label set to be *acceptable* (A) if at least one of the tags in the set describes the image’s content to some extent, even if the name of the target object is not present in it. For example, a potential grid for the target object car may also contain other objects like road, traffic light, and crosswalk. If an image recognition API returns a label set with the tags crosswalk, street, and traffic light, we consider it acceptable even though it does not contain the tag for the main target

object (a “car” in this case). We consider a label set to be an *exact match* (EM) if one of the returned labels in the set matches the name of the target object. A label set is said to be *misclassified* (MC) if none of the returned labels in the label set has any semantic relation with the grid’s actual content. For instance, as shown in Table 6, Google Cloud Vision API misclassified the potential grid number 3, which is an image of a fire hydrant, as an Art (or a Plant). A label set is considered *empty* (E) if the API does not return any label. In our experiment, we find many grids that are misclassified by image recognition services with very high (over 90%) probability, which is a strong indication that those grids are adversarial. Generally, non-adversarial perturbation does not mislead a well-trained image classifier; instead, it degrades the target class’s confidence score. At the same time, vision APIs for image recognition return correct or acceptable label sets for some noisy grids (see Table 7). Hence, based on our findings, we hypothesize that reCAPTCHA is using a mixture of adversarial perturbations and random noises in some challenge images. Note that the actual identification of adversarial perturbations or examples is a non-trivial task, and it is still an open research problem in the AI domain.

Next, we investigate the impact of adversarial perturbations or random noises on our object detection models. We run our pre-trained object detection models on the same 52 images to determine how many target objects they can correctly identify. Note that, it is not always necessary to detect and localize all the target objects in a challenge image, *i.e.*, it is sufficient to pass a challenge if the *bounding box to grid mapping* algorithm result can capture all the potential grids regardless of the number of objects present in the challenge image. Our base object detection models can recognize less than 60% of the target objects in the challenge images (see Table 8). We use the object counts as a metric to assess object detection models’ performance to simplify the analysis.

Next, we apply different data augmentation techniques to study whether retraining the object detection system using the augmented data helps increase the system’s robustness against reCAPTCHA’s anti-recognition mechanisms. We develop a data augmentation pipeline employing various augmentation methods such as additive Gaussian noise injection, blurring, and changing the brightness and contrast, etc. We utilize the *imgaug* [31] library for data augmentation. For adding Gaussian noise, we use the `AdditiveGaussianNoise` function with `scale=(0, .2*255)`. We use the following methods for blurring: `GaussianBlur` with `sigma=(0.5-5.0)`, `MedianBlur` with `k=(5, 17)`, and `AverageBlur` with `k=(5, 17)`. Figure 8 in Appendix B shows some examples of data augmentation methods applied to a sample reCAPTCHA challenge image. We randomly select 30% of training images and apply each data augmentation method from our pipeline to images. Finally, we retrain our object detection models using the augmented training samples. We also collect 500 perturbed images ($\sigma_{noise} > 5$) from the reCAPTCHA challenges and fine-tune the mod-

Table 6: A noisy image from reCAPTCHA challenge and the labels returned by 4 image recognition services. The target object is a “Fire Hydrant.” PGNs=Potential Grid Numbers (1, 3, 4).

Image	PGNs	Google Cloud Vision	Microsoft Azure Computer Vision	Amazon Rekognition	Clarifai
	1	leaf, grass, plant	animal, mammal	art, painting	desktop, animal, dog, people, nature
	3	pattern, art	grass, outdoor	hydrant, fire hydrant	nature, abstract, pattern, art
	4	pink, toy, action figure	hydrant, outdoor object, fire hydrant	hydrant, fire hydrant	travel, old, art, desktop

els further by utilizing *adversarial training*. Note that, even though we call it adversarial training, some training samples may not include adversarial noises. Since reCAPTCHA’s source code and data are not open-source, it is challenging to do further verification.

Table 8 depicts the performance of our object detection models. We can see that the object detection models with augmented data can detect 149 (over 73%) out of 203 target objects in the perturbed images. That is over 17% performance improvement with respect to our base models. It is also evident that adversarial training provides significant performance boosts further while using only 500 training samples. We suspect reCAPTCHA might be generating the perturbation from a simple data distribution, which enabled us to achieve such a great increase in performance despite training against only a small number of adversarial samples. We expect that adding more perturbed images from original reCAPTCHA challenges will further enhance the detection performance. Notice that models have misidentified some objects after performing data augmentation and adversarial training. We can reduce the number of misdetections by setting the detection threshold to a higher value (our default is 0.2). However, doing so slightly degrades the overall performance of the object detection models.

We note that advanced object detection systems, such as YOLOv3, are less susceptible to anti-recognition techniques employed by reCAPTCHA in general. For example, our base object detection models (Table 8) perform much better in identifying objects in the perturbed images than vision APIs for image recognition in Table 7. We assume that reCAPTCHA mainly targeted image recognition and classification systems because all of the prior attacks against reCAPTCHA in literature are based on them. In summary, our findings imply that an effectively trained object detection based solver can neutralize reCAPTCHA’s anti-recognition attempts.

IP address rate-limit. To study whether reCAPTCHA enforces any IP address rate limit, we set up a 3-day experiment. We select 3 reCAPTCHA-enabled websites and attempt to initiate 1000 reCAPTCHA challenges to a chosen website each day. We limit ourselves to 1000 requests to a site each day to minimize the impact on the test website. Further, there

Table 7: Results of noisy grid classification returned by image recognition services. EM=No. of exact match label sets. A=No. of acceptable label sets. E=No. of empty label sets. MC=No. of misclassifications label sets.

Service	# of Grids	Labels			
		EM	A	E	MC
Google Cloud Vision	172	31	68	32	41
Microsoft Azure Computer Vision	172	19	82	9	62
Amazon Rekognition	172	70	58	0	44
Clarifai	172	27	71	0	74

is a delay of 60 seconds between two subsequent requests. We perform this experiment with 3 IP addresses: an institutional IP, a residential IP, and a Tor anonymity network IP.

We experiment with the academic IP first. On the first day, we can initiate 818 reCAPTCHA challenges, and the remaining 182 attempts have been blocked. On the second day, we are able to initiate the reCAPTCHA challenges 801 times, and the remaining 199 attempts have been blocked. On the third day, none of our attempts has been blocked. The duration of the blocking period usually ranges from 36 to 95 minutes. Note that getting the IP blocked in one website by reCAPTCHA does not generally restrict that same IP from initiating reCAPTCHA challenges on other websites, which is normal behavior. We could initiate reCAPTCHA challenges more than 800 times from a single IP address to a particular website in any of the cases. Next, we repeat this experiment from the same machine but with a residential IP and observe a similar pattern. Finally, we experiment by tunneling the traffic through the Tor network with the exit node in Germany (selected randomly by the Tor client). During this experiment, a significant number of requests have been blocked by reCAPTCHA. Specifically, at least 30% of our requests are blocked each day. It is worth noting that the exit node’s geolocation does not usually make that much of a difference. We confirm this by repeating the experiment separately with a manually specified exit node in three different

Table 8: Impact of anti-recognition on object detection models.

Model	Objects	Objects Detected	Objects Detected with Wrong Label
Base (B)	203	114	1
B+Basic Augmentation (BA)	203	149	11
B+BA+Adversarial Training	203	167	13

regions, namely North America (the US), Europe (Netherlands), and Asia (Hong Kong). It implies that reCAPTCHA considers requests originating from an IP within the Tor network to be highly suspicious.

Our findings also indicate that reCAPTCHA’s anti-bot technology follows a relaxed per IP address rate-limit approach towards regular IP addresses. While it may be reasonable for a modern web application to allow thousands of requests from a single client machine with a unique IP address, a webpage dedicated for the sole purpose of user registration or login may not want to provide such freedom. Therefore, we recommend letting the website owners set up a custom daily IP address rate-limit by adding such an option in reCAPTCHA deployment settings and enforcing the restriction from the reCAPTCHA backend.

6 Economic analysis

We use five popular human-based online CAPTCHA solving services to compare their performance with our system. The services are 2Captcha [1], Anti-Captcha [3], BestCaptchaSolver [4], DeathByCaptcha [6], and Imagetyperz [8]. We submit 500 reCAPTCHA challenges to each service, totaling 2500 challenges for all the services combined. The average success rate and speed of breaking reCAPTCHA challenges are shown in Table 9. As we can see, our system outperforms both BestCaptchaSolver and Imagetyperz. While 2Captcha, Anti-Captcha, and DeathByCaptcha perform a little better than ours, they are significantly slower. Our system is more than 3.5x faster than the fastest human-based CAPTCHA solving service. Further, the adversaries can run our system with virtually no cost by deploying it on their machines. Overall, our system’s performance is comparable to that of human-based online CAPTCHA solving services, and scammers could use it as an alternative to human workers to automatically solve reCAPTCHA challenges.

7 Comparing to prior attacks

Sivakorn *et al.* leveraged online image annotation APIs to break the earlier implementation of reCAPTCHA (reCAPTCHA 2015) with a success rate of 70.78% [44]. Since then, reCAPTCHA has changed significantly. In a similar attack, Weng *et al.* evaluated the security of 10 real-world

Table 9: Performance of human-based CAPTCHA solving services.

Service	Success Rate (%)	Speed (s)
2Captcha	98.2	73.11
Anti-Captcha	92.4	83.99
BestCaptchaSolver	67.2	93.42
DeathByCaptcha	96.2	78.33
Imagetyperz	73	131.4
Our system	83.25	19.93

image CAPTCHAs, including reCAPTCHA 2018 [50]. They used a CNN-based image classification model to break reCAPTCHA 2018 challenges with a success rate of 79%. Note that reCAPTCHA 2018 used to show relatively simple images when compared to the current reCAPTCHA challenges. Further, Weng *et al.* encountered only 10 image categories in the reCAPTCHA 2018 challenges, where we come across 18 object categories in the latest version of reCAPTCHA. Moreover, the anti-recognition mechanism employed by the current reCAPTCHA was not available in reCAPTCHA 2018 as well.

In summary, we propose a new approach to breaking the most advanced version of reCAPTCHA using object detection models. Our method significantly outperforms prior approaches as well as off-the-shelf object detection APIs. We believe the stark difference in the performance between our solver and off-the-shelf object detection APIs is because we train our solver to handle reCAPTCHA object categories exclusively. In contrast, object detection APIs are developed for general-purpose object detection tasks. Further, as discussed before, we assume that these services are still in their early development stages.

8 Discussion

8.1 Ethics

We did not affect the security or the availability of the tested websites during our data collection for preliminary analysis or performing a live attack on reCAPTCHA as we limit our access within the two `iframe` elements related to the challenge. We also disclosed our findings to Google when we developed our system’s initial implementation in August 2019. Unfortunately, we have not noticed any discernible changes to reCAPTCHA by Google that can prevent our attack. Our system can still break the reCAPTCHA challenges with a high success rate as of March 2020.

We have not published the source code of our tool due to concerns over potential abuse by scammers and fraudsters alike. However, we encourage researchers to contact us if they want to use our tool for research purposes only.

8.2 Limitation

We design our attack to break reCAPTCHA challenges specifically. While reCAPTCHA is the most widely deployed CAPTCHA service on the Web, there are other popular image CAPTCHA schemes. It will be interesting to see if we could extend our object detection based solver module to attack a whole family of similar image CAPTCHA designs. We plan to conduct a study on the generalization of our attack as a future extension.

8.3 Countermeasures

While it may not be possible to prevent our attack completely, we provide several countermeasures to limit it.

Content heterogeneity. Our experiment shows that content homogeneity has contributed to lowering the accuracy of image recognition and classification services. However, it has minimal to no impact on our object detection based solver. As such, reCAPTCHA's current approach to resisting automated attacks does not seem to be working. We recommend using images from diverse and heterogeneous sources, which will provide the CAPTCHA designers more flexibility if they need to expand the total number of object categories.

Incorporate natural language understanding to image CAPTCHA test. The natural language understanding is considered as one of the three biggest open problems in natural language processing [14]. This weakness could be exploited to strengthen the security of image CAPTCHA. We suggest utilizing the natural language understanding in forming the challenge instruction so that the direction needed to solve a challenge must be inferred through natural language reasoning. The current design of reCAPTCHA makes this information readily available to the attacker.

Use spatial properties of the object. The main design flaw of reCAPTCHA is that an advanced object detection system can solve its underlying AI problem for telling humans and bots apart. The problem could be hardened for the machine by exploiting the object's spatial attributes, such as shape, size, orientation, tilt direction, etc. However, it may require extensive research to determine whether designing such a CAPTCHA scheme is feasible in practice.

9 Related work

CAPTCHA is an active research area, and there exists an extensive body of studies in this area. Due to space limitations, we only discuss the works that are mostly related to ours. Further, we mainly focus on CAPTCHA attack related research.

Image CAPTCHAs. Golle *et al.* [29] used support vector machine classifiers to break Asirra CAPTCHA [26]. Zhu *et al.* analyzed the security of various earlier image CAPTCHAs and proposed attacks to break them [54]. Sivakorn *et al.* used

deep learning techniques to break reCAPTCHA 2015 [44]. Later Weng *et al.* analyzed the security of several real-world image CAPTCHAs, including reCAPTCHA 2018, and developed deep learning-based attacks that succeeded in breaking all the CAPTCHAs tested in their work [50]. Osadchy *et al.* proposed a new CAPTCHA scheme called DeepCAPTCHA that exploits adversarial examples in CAPTCHA image generation to deceive DNN image classifiers [37]. Shi *et al.* proposed a framework for generating text and image adversarial CAPTCHAs [43].

Text CAPTCHAs. Most text CAPTCHA schemes have been broken [20,28,34,35,51,52]. Chellapilla *et al.* proposed using machine learning algorithms to break earlier text CAPTCHA designs [20]. Yan *et al.* used simple pattern recognition algorithms to break most of the text CAPTCHAs provided at Captchaservice.org with a near-perfect success rate [51]. El Ahmad *et al.* proposed a novel attack against reCAPTCHA v1 2010 [25]. In 2011, Bursztein *et al.* evaluated the security of 15 CAPTCHA schemes from popular web sites and concluded that 13 of them were vulnerable to automated attacks [19]. In 2014, Bursztein *et al.* used a machine learning-based generic attack to break many popular real-world text CAPTCHA schemes, including reCAPTCHA 2011 and reCAPTCHA 2013 [17]. In 2016, Gao *et al.* were able to break many text CAPTCHAs using a low-cost attack that uses Log-Gabor filters [28]. In 2018, Ye *et al.* proposed a Generative Adversarial Networks (GANs) based approach to break 33 text CAPTCHA schemes [53].

Audio CAPTCHAs. Audio CAPTCHAs designed as alternative CAPTCHA schemes for visually impaired users have been subjected to automated attacks over the years [16,18,41,45,47]. Tam *et al.* analyzed the security of audio CAPTCHAs from popular websites by using machine learning techniques and were able to break many of them, including an earlier version reCAPTCHA [47]. In 2017, Bock *et al.* proposed an automated system called unCaptcha that could break reCAPTCHA's audio challenges with an accuracy above 85% [16]. In 2017, Solanki *et al.* proposed a low-cost attack against several popular audio CAPTCHAs using off-the-shelf speech recognition services [45].

10 Conclusion

CAPTCHAs have become a standard security mechanism for protecting websites from abusive bots. In this work, we showed that one of the Internet's most widely used image CAPTCHA schemes, reCAPTCHA v2, could be broken by an automated solver based on object detection method with a high success rate. Our extensive analysis showed that despite several major security updates to counter automated attacks, which could invalidate prior image recognition and classification based solvers, reCAPTCHA remains vulnerable to advanced object detection systems. Given the capabilities of the current object detection systems, we think that reCAPTCHA

is essentially broken because its reverse Turing tests to distinguish humans from bots are easily solvable by an object detection based automated solver.

Acknowledgment

This work is partially supported in part by US NSF under grants OIA-1946231. The work in China is supported by Cooperation and Exchange Program of International Science and Technology of Shaanxi Province (2019KW-010). We also want to thank our shepherd Kevin Borgolte and the anonymous reviewers for valuable comments.

References

- [1] 2captcha. <https://2captcha.com/>. Last accessed 12 January 2020.
- [2] Amazon rekognition. <https://aws.amazon.com/rekognition/>. Last accessed 22 December 2020.
- [3] Anti captcha. <https://anti-captcha.com/mainpage>. Last accessed 12 January 2020.
- [4] Best captcha solver. <https://bestcaptchasolver.com/>. Last accessed 12 January 2020.
- [5] Clarifai. <https://www.clarifai.com/>. Last accessed 22 December 2020.
- [6] Death by captcha. <https://www.deathbycaptcha.com/>. Last accessed 12 January 2020.
- [7] Google cloud vision api. <https://cloud.google.com/vision>. Last accessed 22 December 2020.
- [8] Imagetyperz. <http://www.imagetyperz.com/>. Last accessed 12 January 2020.
- [9] Microsoft azure computer vision. <https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/>. Last accessed 22 December 2020.
- [10] puppeteer-firefox. <https://github.com/puppeteer/puppeteer/tree/master/experimental/puppeteer-firefox>. Last accessed 22 July 2019.
- [11] reCAPTCHA Usage Statistics. <https://trends.builtwith.com/widgets/reCAPTCHA-v2>. Last accessed 21 July 2019.
- [12] Selenium - Web Browser Automation. <https://selenium.dev/>. Last accessed 20 December 2019.
- [13] COCO - Common Objects in Context. <http://cocodataset.org/#overview>, 2019. Last accessed 8 Aug 2019.
- [14] Deep Learning Indaba 2018. Frontiers of natural language processing. https://www.kamperh.com/slides/ruder+kamper_indaba2018_talk.pdf, 2018.
- [15] Battista Biggio, Iginio Corona, Davide Maiorca, Blaine Nelson, Nedim Šrđić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. *Lecture Notes in Computer Science*, page 387–402, 2013.
- [16] Kevin Bock, Daven Patel, George Hughey, and Dave Levin. uncaptcha: A low-resource defeat of recaptcha’s audio challenge. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT’17, Berkeley, CA, USA, 2017. USENIX Association.
- [17] Elie Bursztein, Jonathan Aigrain, Angelika Moscicki, and John C. Mitchell. The end is nigh: Generic solving of text-based captchas. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT’14, Berkeley, CA, USA, 2014. USENIX Association.
- [18] Elie Bursztein and Steven Bethard. Decaptcha: Breaking 75% of ebay audio captchas. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT’09, Berkeley, CA, USA, 2009. USENIX Association.
- [19] Elie Bursztein, Matthieu Martin, and John Mitchell. Text-based captcha strengths and weaknesses. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS ’11, page 125–138, New York, NY, USA, 2011. Association for Computing Machinery.
- [20] Kumar Chellapilla and Patrice Y. Simard. Using machine learning to break visual human interaction proofs (hips). In *Proceedings of the 17th International Conference on Neural Information Processing Systems*, NIPS’04, pages 265–272, Cambridge, MA, USA, 2004. MIT Press.
- [21] Thomas Claburn. Google’s recaptcha favors – you guessed it – google: Duh, only a bot would refuse to sign into the chocolate factory. https://www.theregister.co.uk/2019/06/28/google_recaptcha_favoring_google/, 2019. Last accessed 8 August 2019.

- [22] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks, 2016.
- [23] Google Developers. Choosing the type of recaptcha. <https://developers.google.com/recaptcha/docs/versions#v1>, 2019. Last accessed 23 July 2019.
- [24] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association.
- [25] Ahmad S El Ahmad, Jeff Yan, and Mohamad Tayara. The robustness of google captchas. Technical report, School of Computer Science, Newcastle University, UK, May 2011.
- [26] Jeremy Elson, John Douceur, Jon Howell, and Jared Saul. Asirra: A captcha that exploits interest-aligned manual image categorization. pages 366–374, 01 2007.
- [27] Haichang Gao, Wei Wang, Jiao Qi, Xuqin Wang, Xiyang Liu, and Jeff Yan. The robustness of hollow captchas. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1075–1086, New York, NY, USA, 2013. Association for Computing Machinery.
- [28] Haichang Gao, Jeff Yan, Fang Cao, Zhengya Zhang, Lei Lei, Mengyun Tang, Ping Zhang, Xin Zhou, Xuqin Wang, and Jiawei Li. A simple generic attack on text captchas. In *NDSS*, 2016.
- [29] Philippe Golle. Machine learning attacks against the asirra captcha. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 535–542, New York, NY, USA, 2008. ACM.
- [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015.
- [31] Alexander B. Jung, Kentaro Wada, Jon Crall, Satoshi Tanaka, Jake Graving, Christoph Reinders, Sarthak Yadav, Joy Banerjee, Gábor Vecsei, Adam Kraft, Zheng Rui, Jirka Borovec, Christian Vallentin, Semen Zhydenko, Kilian Pfeiffer, Ben Cook, Ismael Fernández, François-Michel De Rainville, Chi-Hung Weng, Abner Ayala-Acevedo, Raphael Meudec, Matias Laporte, et al. imgaug. <https://github.com/aleju/imgaug>, 2020. Online; accessed 01-Feb-2020.
- [32] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
- [33] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.
- [34] Greg Mori and Jitendra Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'03, pages 134–141, Washington, DC, USA, 2003. IEEE Computer Society.
- [35] Gabriel Moy, Nathan Jones, Curt Harkless, and Randall Potter. Distortion estimation techniques in solving visual captchas. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, pages 23–28, Washington, DC, USA, 2004. IEEE Computer Society.
- [36] neuroradiology. Insiderecaptcha. <https://github.com/neuroradiology/InsideReCaptcha>, 2014. Last accessed 8 August 2019.
- [37] M. Osadchy, J. Hernandez-Castro, S. Gibson, O. Dunkelmann, and D. Pérez-Cabo. No bot expects the deep-captcha! introducing immutable adversarial examples, with applications to captcha generation. *IEEE Transactions on Information Forensics and Security*, 12(11):2640–2653, Nov 2017.
- [38] Joseph Redmon. Darknet: Open source neural networks in c. <http://pjreddie.com/darknet/>, 2013–2016. Last accessed 21 July 2019.
- [39] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.
- [40] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, Jun 2017.
- [41] Shotaro Sano, Takuma Otsuka, Katsutoshi Itoyama, and Hiroshi Okuno. Hmm-based attacks on google's recaptcha with continuous visual and audio symbols. *Journal of Information Processing*, 23:814–826, 11 2015.
- [42] Katharine Schwab. Google's new recaptcha has a dark side. <https://www.fastcompany.com/90369697/googles-new-recaptcha-has-a-dark-side>, 2019. Last accessed August 2019.

- [43] Chenghui Shi, Xiaogang Xu, Shouling Ji, Kai Bu, Jianhai Chen, Raheem Beyah, and Ting Wang. Adversarial captchas, 2019.
- [44] Suphanee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. I am robot: (deep) learning to break semantic image captchas. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Mar 2016.
- [45] Saumya Solanki, Gautam Krishnan, Varshini Sampath, and Jason Polakis. In (cyber)space bots can hear you speak: Breaking audio captchas using ots speech recognition. pages 69–80, 11 2017.
- [46] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013.
- [47] Jennifer Tam, Jiri Simsa, Sean Hyde, and Luis V. Ahn. Breaking audio captchas. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1625–1632. Curran Associates, Inc., 2009.
- [48] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.
- [49] Luis von Ahn, Manuel Blum, and John Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, February 2004.
- [50] Haiqin Weng, Binbin Zhao, Shouling Ji, Jianhai Chen, Ting Wang, Qinming He, and Raheem Beyah. Towards understanding the security of modern image captchas and underground captcha-solving services. *Big Data Mining and Analytics*, 2:118–144, 06 2019.
- [51] Jeff Yan and Ahmad Salah El Ahmad. Breaking visual captchas with naive pattern recognition algorithms. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 279–291, 2007.
- [52] Jeff Yan and Ahmad Salah El Ahmad. A low-cost attack on a microsoft captcha. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS ’08, pages 543–554, New York, NY, USA, 2008. ACM.
- [53] Guixin Ye, Zhanyong Tang, Dingyi Fang, Zhanxing Zhu, Yansong Feng, Pengfei Xu, Xiaojiang Chen, and Zheng Wang. Yet another text captcha solver: A generative adversarial network based approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, pages 332–348, New York, NY, USA, 2018. ACM.
- [54] Bin B. Zhu, Jeff Yan, Qiuji Li, Chao Yang, Jia Liu, Ning Xu, Meng Yi, and Kaiwei Cai. Attacks and design of image recognition captchas. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS’10, pages 187–200, New York, NY, USA, 2010. ACM.

Appendix A Captcha Types

Figure 6 shows an example of a selection-based CAPTCHA challenge, and Figure 7 shows an example of a click-based CAPTCHA challenge.

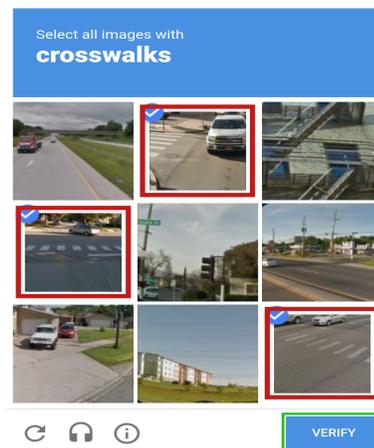


Figure 6: Selection-based image CAPTCHA.

Appendix B Data Augmentation

Figure 8 depicts some examples of data augmentation methods applied to a sample reCAPTCHA challenge image.

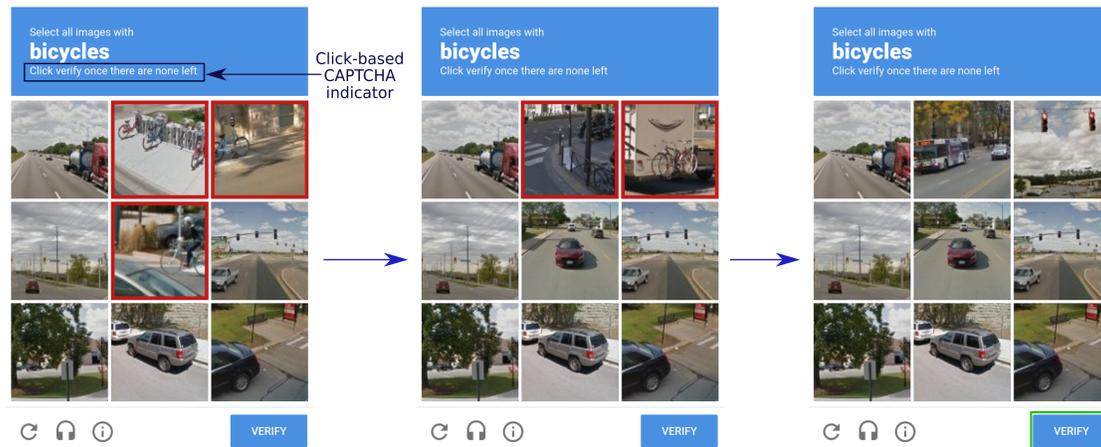


Figure 7: Click-based image CAPTCHA.

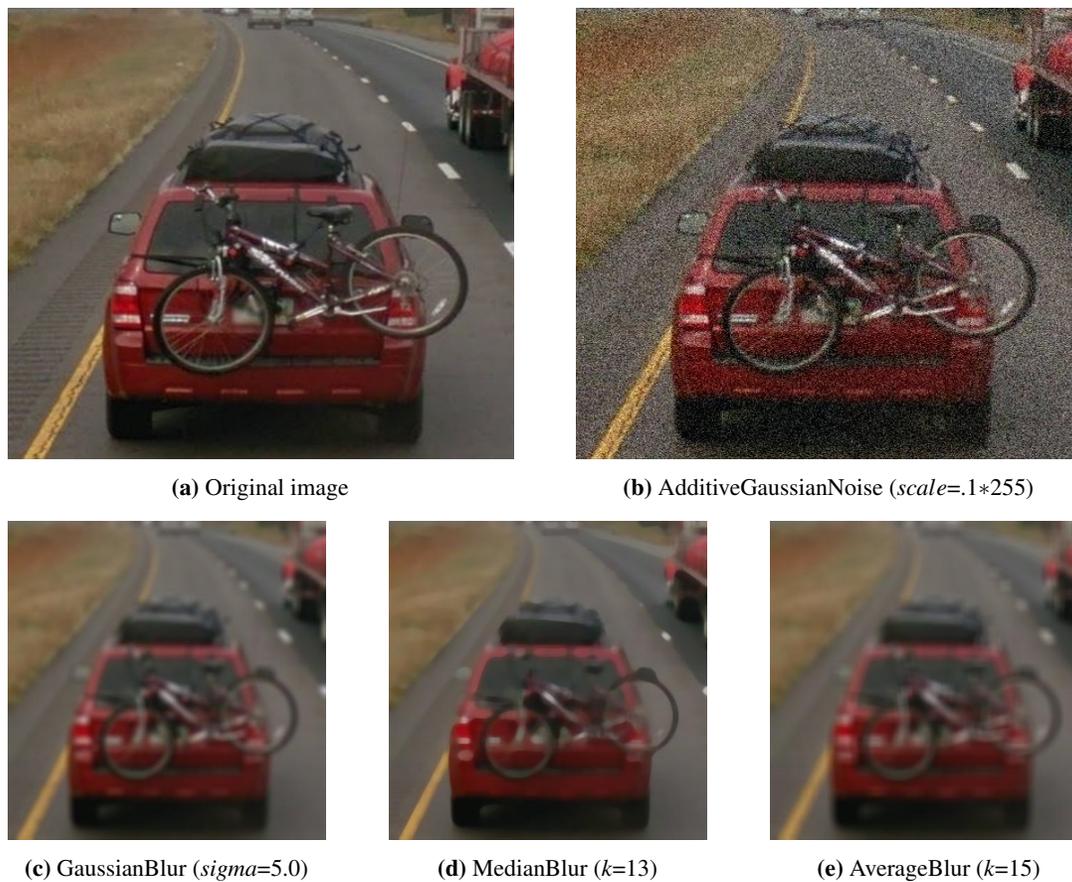


Figure 8: Examples of data augmentation methods.

Evasion Attacks against Banking Fraud Detection Systems

Michele Carminati, Luca Santini, Mario Polino, and Stefano Zanero

Politecnico di Milano

{michele.carminati, mario.polino, stefano.zanero}@polimi.it

luca2.santini@mail.polimi.it

Abstract

Machine learning models are vulnerable to adversarial samples: inputs crafted to deceive a classifier. Adversarial samples crafted against one model can be effective also against related models. Therefore, even without a comprehensive knowledge of the target system, a malicious agent can attack it by training a surrogate model and crafting evasive samples. Unlike the image classification context, the banking fraud detection domain is characterized by samples with few aggregated features. This characteristic makes conventional approaches hardly applicable to the banking fraud context.

In this paper, we study the application of Adversarial Machine Learning (AML) techniques to the banking fraud detection domain. To this end, we identify the main challenges and design a novel approach to perform evasion attacks. Using two real bank datasets, we evaluate the security of several state-of-the-art fraud detection systems by deploying evasion attacks with different degrees of attacker's knowledge. We show that the outcome of the attack is strictly dependent on the target fraud detector, with an evasion rate ranging from 60% to 100%. Interestingly, our results show that the increase of attacker knowledge does not significantly increase the attack success rate, except for the full knowledge scenario.

1 Introduction

Nowadays, machine learning techniques are applied to several data-driven tasks: from image identification [22], face recognition [30], and natural language processing [33] to malware [3, 27], and intrusion detection [28]. Machine learning has gained importance also in the banking fraud detection domain [6, 11, 18, 26]. Those systems present prominent benefits, but, unfortunately, they suffer from the typical weakness of machine learning models: it is possible to significantly reduce their robustness and alter their performance through *adversarial samples* [8, 19, 25]. Adversarial samples are inputs crafted starting by legitimate samples that are iteratively perturbed to make the detector to misclassify them. There are many

studies on how to craft adversarial samples that propose different approaches based on the gradient computation, which have advantages and drawbacks depending on the domain of application. There are also many works about *transferability*, the property that captures the ability of an attack against a machine learning model to be effective against a different, potentially unknown, model [17, 23]. This property allows a malicious user to design an attack against a machine learning-based system also in the case in which he does not know the target system [16, 24]. AML has been studied mainly in the field of image classification in which, due to some intrinsic characteristics of images, researchers obtained remarkable results. In recent years, many studies applied AML to other fields such as malware detection, where researchers had to overcome the challenges of that domain [3, 20].

In this paper, we study the application of AML techniques to the banking fraud detection domain, which, unlike the image classification one, is characterized by samples with few aggregated features. This characteristic makes conventional approaches hardly applicable to this context. Ergo, we present a novel approach to perform evasion attacks against banking fraud detection systems that adapts and extends some existent methods for crafting adversarial samples and mitigate the challenges of the fraud detection domain. We study different threat models characterized by attackers with different degrees of knowledge: **Black-Box**, with zero knowledge of the target system; **Gray-Box**, with partial knowledge of the system; **White-Box**, with complete knowledge of the system. Using two real anonymized bank datasets with only legitimate transactions, we show how a malicious attacker can compromise state-of-the-art banking fraud detection systems by deploying evasion attacks, with an evasion rate ranging from 60% to 100%. The contributions are the following:

- We present a novel machine learning-based approach to perform evasion attacks against fraud detection methods under different degrees of knowledge and simulating the behavior of different types of fraudsters.
- We study the application of state-of-the-art AML algo-

cific class or as any of the classes of interest. In a binary classification problem, like the fraud detection one, the attacker wants the detector to classify a malicious transaction (i.e., frauds) as legitimate or vice versa.

In our threat model, the attacker performs an evasion attack, which is an *integrity* violation that can be generic or targeted. The attacker carries out malicious transactions on behalf of the user and wants the detector to classify them as legitimate.

3.2 Attacker’s Knowledge

We modeled the attacker with different degrees of knowledge, extending the characterization of Biggio *et al.* [5], by adding a term representing the knowledge the attacker has concerning the past transactions of users. In fact, in the fraud detection domain, we need to extract aggregated features to capture user behavior. To perform transaction aggregation, an attacker needs the last transactions of the user. In our approach, the adversary needs just one month of the target victim’s transaction history. Regarding our dataset, one month of transactions corresponds to observe an average of 18 ± 28 transactions per victim. The high standard deviation is due to the fact that our dataset is highly skewed¹, with the majority of users performing few transactions. For example, in a real case, the attacker retrieves the victim’s transaction history from the banking application, thanks to a malware injected into the victim’s devices. We use the following terms to describe attacker’s knowledge: training data \mathcal{D} ; feature set \mathcal{X} (i.e., how to aggregate samples); learning algorithm along with the loss function to minimize f ; trained parameters/hyper-parameters w ; past data (i.e., transactions) of the victim user \mathcal{H} . In summary, the attacker’s knowledge can be characterized by the tuple $\Theta = (\mathcal{D}, \mathcal{X}, f, w, \mathcal{H})$ ². Based on the previous assumptions, we can configure four scenarios:

White-Box. The attacker is assumed to know everything. For example, he or she is an intern at the bank and collected all the information. Even if this is a strong assumption, it is advantageous to perform a worst-case evaluation of the fraud detectors. It also provides an upper bound that we can use to compare to other more restrictive scenarios. The tuple describing this setting is $\Theta_{wb} = (\mathcal{D}, \mathcal{X}, f, w, \mathcal{H})$ ¹.

Gray-Box. The attacker has partial knowledge about the detection system. In particular, he/she knows how the fraud detector aggregates data to compute the features (\mathcal{X}) but not the training data, the learning algorithm, and the trained hyper-parameters ($\hat{\mathcal{D}}, \hat{f}, \hat{w}$). As motivated before, we assume that the attacker has retrieved one month of past transactions ($\tilde{\mathcal{H}}$) to compute an estimation of the aggregated features. The tuple describing this setting is $\Theta_{gb} = (\hat{\mathcal{D}}, \mathcal{X}, \hat{f}, \hat{w}, \tilde{\mathcal{H}})$ ¹.

¹By considering only users with more than 5 monthly transactions, statistics about the number of transactions per user are the following: $\mu = 18.41$, $\sigma = 27.91$, $\sigma^2 = 779$, $median = 11$, $mode = 6$, $Q_1 = 7$, $Q_3 = 19$.

²The term x indicates a full knowledge of term x , \tilde{x} indicates a partial knowledge of term x , and \hat{x} indicates zero knowledge of x .

Black-Box with Data. The attacker has no knowledge about the detection system ($\hat{\mathcal{X}}, \hat{f}, \hat{w}$), but he/she has at his/her disposal the same training set used by the target system ($(\mathcal{D}$ and $\mathcal{H})$ consisting of all the banking transactions of the last few months. Thank to this dataset, the attacker can compute a precise estimate of the aggregated features. The tuple describing this setting is $\Theta_{bb} = (\mathcal{D}, \hat{\mathcal{X}}, \hat{f}, \hat{w}, \mathcal{H})$ ¹.

Black-Box. The attacker has no knowledge about the detection system and training data ($\hat{\mathcal{D}}, \hat{\mathcal{X}}, \hat{f}, \hat{w}$). However, as motivated before, we assume that the attacker has retrieved one month of past transactions ($\tilde{\mathcal{H}}$) to compute an approximation of the aggregated features. Also, the attacker has at his/her disposal a set of transactions different from the ones used by the target detection system (i.e., an old leaked dataset or a dataset belonging to another financial institution). The tuple describing this setting is $\Theta_{bb} = (\hat{\mathcal{D}}, \hat{\mathcal{X}}, \hat{f}, \hat{w}, \tilde{\mathcal{H}})$ ¹.

3.3 Attacker’s Capability

This characteristic highlights the power of the attacker concerning the system. It outlines the **attacker’s influence** of manipulating data and the domain-specific **data manipulation constraints**. If the attacker can manipulate only the test set, the attack is called *exploratory* or *evasion*. If the attacker can manipulate both the test set and the training set, the attack is called *causative* or *poisoning*. In this work, we focus on evasion attacks. Therefore, the attacker can manipulate the test set and execute transactions on behalf of the user. Then, the fraud detection system will process these transactions following the procedure described in Figure 1. State-of-the-art fraud detectors extract information from the past user’s transactions to compute aggregated features that capture the user’s spending behavior. As anticipated in Section 2, the attacker is, therefore, subjected to a strong limitation: he or she can directly modify only some features, while others also depend on the past behavior of the user. We go deeper into this concept in Appendix B.1. As anticipated in Section 3.2, our approach needs just a short transaction history (one month) of the target victim and the possibility to perform transactions. There are several ways to recover the data to perform an evasion attack. The attacker can infect bank customers’ device with a Trojans (e.g., Zeus, Citadel). Alternatively, the attacker can use phishing techniques to retrieve the victim’s bank access credentials and One Time Password (OTP). At this point, the attacker can obtain the transactions carried out by the user and execute them on his or her behalf. The attacker retrieves the transaction history of the victim from the banking application and computes aggregated features (e.g., total money spent in one month, average daily money spent). With this information, the attacker performs one or more evasive transactions without being detected. A careful reader will notice that transaction history can also be obtained by passively observing a user with a persistent malware sample for a while.

4 Datasets Analysis and Engineering

Our dataset comes from an important banking group; we worked on two real datasets that have been thoroughly inspected and cleaned from frauds. Data is anonymized by removing personal information related to users and replaced with hashed values that preserve equality. In Table 1, we show the number of users and transactions for each dataset. The most relevant features are: the transaction amount (**Amount**); the *hashed* unique ID associated to the user (**UserID**); the date and the time of the execution of the transaction (**Timestamp**); the *hashed* IP Address of the connection from which the transaction is performed (**IP**); the *hashed* International Bank Account Number of the beneficiary (**IBAN**); the country code of the beneficiary IBAN (**IBAN_CC**); the country code and the autonomous system number from which the connection comes (**CC_ASN**); the identifier of a session (**SessionID**).

4.1 Feature Extraction Strategies

To design a good fraud detection algorithm, a feature selection and extraction strategy need to be chosen [1, 15, 31]. A strategy, which captures the user spending pattern, is a combination of direct derivable features (e.g., amount, country) and aggregated ones (e.g., average, total). In practice, it consists of grouping transactions by features and, then, computing aggregated metrics. First, we selected the direct derivable features and aggregated features used in previous works [1, 9, 12, 32, 35]. Then, with the support of the banking domain experts, we combine them using different strategies that capture different aspects of user spending profiles. Finally, for each fraud detector algorithm considered in this work, we selected the strategies that performed best with our data in detecting synthetic frauds (see Section 4.2 using the holdout validation method). In particular, we empirically evaluated each strategy by using a wrapper approach [21] that maximizes the True Positives (TPs) and minimizes False Positives (FPs). Table 2 summarizes the three best strategies – we will refer to them as **A**, **B**, and **C** – that we selected for the fraud detectors. For an exhaustive description of each feature and aggregation function, we refer the reader to Appendix A.

4.2 Dataset Augmentation: Synthetic Frauds

To train the supervised learning models considered in this paper, we enrich datasets with synthetic frauds generated thanks

Table 1: Number of transactions and users for each dataset

DATASET	USERS	TRANSACTIONS	TIME INTERVAL
2012-2013	53,243	548,833	01/2012 - 08/2013
2014-2015	58,481	470,801	10/2014 - 02/2015

to the collaboration with domain experts and based on fraud scenarios that replicate typical real attacks performed against online banking users. In particular, we focus on most spread malicious schemes, which are driven by banking Trojans or phishing: information stealing and transaction Hijacking. In the information-stealing scheme, the attacker exploits a phishing campaign or a Trojan that infects the victim device to steal the credentials and the OTP that the victim inserts in the web pages of the targeted bank. The attacker can then use the stolen credentials to perform transactions on behalf of the victim. The connection comes from the attacker device, not from the victim one. In the transaction hijacking scheme, the Trojan infects the victim device and hijacks legitimate bank transfers made by the user. The main challenge of such an attack is that connections come from the victim’s device (i.e., from the same IP, Session ID, and ASN). Typically, a fraudster may act according to different strategies: he or she may execute a single transaction with a high amount or multiple transactions with lower amounts. Therefore, to define these fraud patterns, we use three variables: **Total Amount**, the target amount that the attacker wants to steal; **Number of Frauds**, the number of frauds in which the attacker wants to divide the attack; **Duration of the Attack**, the total duration time of the attack. So, for example, if the attacker performs an attack with $\text{Total_Amount} = \text{€}10,000$, $\text{Number_of_Frauds} = 24$, $\text{Duration} = 1$ day, he carries out one fraud of about €417 per hour for one day. Using different combinations of the values of these three variables and the two schemes described above, we inject frauds in the dataset by randomly selecting the victims. The banking datasets are known to be extremely unbalanced, usually containing from 0.1% [31] to 1% [11] of frauds. Therefore, common oversampling and undersampling techniques are used to overcome this problem. In this work, we generate about 1% of frauds.

5 Approach

We generate evasive transactions by exploiting an Oracle that predicts the anomaly of a transaction. The Oracle is not 100% accurate, and its performance depends on the degree of the attacker’s knowledge (see Section 3.2). We can use this Oracle to generate candidate transactions that will likely not be considered fraudulent by the targeted fraud detector. As shown in Figure 2, our approach is composed of two phases: training phase, in which we train the Oracle, and a runtime phase, in which we generate the evasive transactions. During the training phase, we train the Oracle by aggregating historical transactions, as described in Section 4.1. In particular, the Oracle is a surrogate fraud detector that models the spending behaviors of users. It is based on a machine learning model that is used in the runtime phase to classify the candidate transactions that the attacker wants to perform. During the runtime phase, depending on the attacker’s knowledge, he or she observes and collects the transactions the victim carries

Table 2: Summary of the features used in our models. Direct features are data not aggregated. Data is aggregated with several time-frames; i.e., ‘mean30d’ is the mean computed over 30 days while ‘mean7d’ is the mean computed over 7 days

STRATEGY	DIRECT FEATURES	AGGREGATED FEATURES
A	amount	count1h, count1d, count30d, sum1h, sum1d, sum30d, iban_count1h, iban_count1d, iban_count30d, iban_sum1h, iban_sum1d, iban_sum30d, ip_count1h, ip_count1d, ip_count30d, ip_sum1h, ip_sum1d, ip_sum30d, distance_from_mean, iban_distance_from_mean, ip_distance_from_mean, mean, iban_mean, ip_mean, is_new_iban, is_new_ip.
B	amount, time_x, time_y, is_national_iban, is_national_asn, operation_type, confirm_sms	count7d, count30d, mean7d, mean30d, std7d, std30d, ip_count7d, ip_count30d, ip_sum7d, ip_sum30d, iban_cc_count7d, iban_cc_count30d, iban_cc_sum7d, iban_cc_sum30d, asn_count7d, asn_count30d, asn_sum7d, asn_sum30d, count, mean, std, count_iban, mean_iban, count_session, mean_session, is_new_iban.
C	amount, time_x, time_y, is_national_iban, is_international, confirm_sms	count1d, sum1d, mean1d, count7d, sum7d, mean7d, count30d, sum30d, mean30d, iban_count1d, iban_count7d, iban_count30d, iban_cc_count1d, iban_cc_count7d, iban_cc_count30d, ip_count1d, ip_count7d, ip_count30d, asn_count1d, asn_count7d, asn_count30d, mean, iban_count, iban_mean, session_count, is_new_iban, is_new_iban_cc, is_new_ip, is_new_cc_asn.

out. Based on the information collected, the attacker generates raw candidate transactions that are aggregated and given in input to the Oracle. The Oracle classifies each candidate transaction, and if it labels it as a fraud, we discard it; otherwise, we use it in the evasion attack by injecting it in the user banking activity.

Assumptions. Our approach is based on three assumptions:

- **Assumption I** The attacker has a dataset of banking transactions for training the Oracle (e.g., an old dataset belonging to the same or a different bank).
- **Assumption II** The attacker can retrieve or observe the transactions carried out by the victim and the funds availability.
- **Assumption III** The attacker can execute transactions on behalf of the victim.

The first assumption is necessary because the attacker needs a bank dataset to train the Oracle. In general, we can state that if the dataset has been obtained from the same bank against which the attack is carried out, the attack reaches better performances. As explained in Section 2, the second assumption is necessary because, depending on the attacker’s knowledge, to process a single transaction, it is necessary to aggregate the previous ones. Hence, the attacker needs to obtain the victim’s transactions. Moreover, the attacker needs to know the funds availability and if the victim is making new transactions during the attack so that he or she is always up to date and can adequately manage those events. The last assumption is necessary to ensure that the attacker can carry out fraud in the real banking system. From a feasibility point of view: the first assumption is more difficult to satisfy because banks rarely release their data publicly, while the

second and third assumptions can be satisfied with a banking Trojan [14].

5.1 Candidate Transaction Generation

Algorithm 1 Find Timestamps. X is the list of timestamps in which the user has performed a transaction, F is the list of time windows sizes, ϵ is a small time delta greater than zero used to fall in the time window

```

1: procedure FINDTIMES( $X, F$ )
2:    $T \leftarrow []$  ▷  $T$  initially empty list
3:   for  $f$  in  $F$ ,  $x$  in  $X$  do
4:      $w_{start} \leftarrow x$ 
5:      $w_{end} \leftarrow w_{start} + f$ 
6:      $T.append(w_{start} + \epsilon, w_{end} + \epsilon)$  ▷  $\epsilon < \min(F)$ 
7:   end for
8:   return  $T$ 
9: end procedure

```

To generate candidate transactions, we efficiently explore the space of possible values of each transaction feature. In truth, the raw candidate transaction features that the attacker can directly control are only the timestamp and the amount. We assume that *IBAN* and *IBAN_CC* refer to the IBAN of the malicious recipient (i.e., new *IBAN* never seen in other transaction) while the *IP* and the *ASN* depend on the attacker strategy. Therefore, the attacker has to choose the amount to steal at each transaction. The choice of the value depends on the strategy chosen by the attacker: conservative or risky. However, a too high amount would lead the Oracle to classify the transaction as fraud, a too low amount does not allow the attacker to maximize the profit. In Section 6, we exhaustively compare different strategies characterized by high, medium,

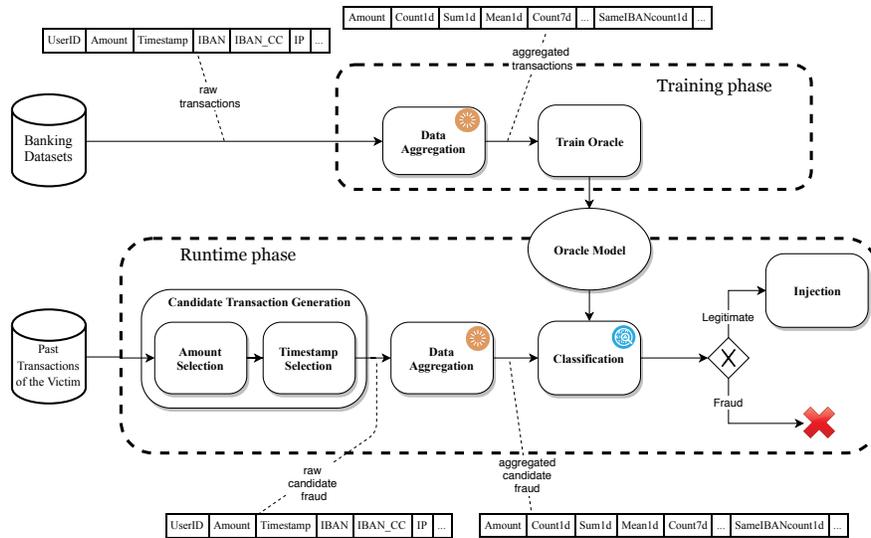


Figure 2: Approach Overview

and low amounts. Once the attacker has selected the amount to steal, he/she identifies the best moment to perform the transaction (i.e., timestamp). A naive approach would be to explore all the possible timestamps within the period in which the attack occurs, but this would lead to a massive number of transactions. To avoid this issue, we bucketize (i.e., group) the timestamps based on the historical transactions using the algorithm described in Pseudocode 1. This procedure takes in input the list of selected time windows sizes and the timestamps in which the user has performed a transaction – the attacker has retrieved them from the victim’s historical transactions. Then, for each time-windows size and timestamp, it builds the aggregated time windows in which the customer is active. The time windows have the objective of capturing the user’s short-term, mid-term, and long-term behavior. We look for the most used sizes in literature [9, 32]: We use one hour and one day for the short-term, seven days for the mid-term, and one month for the long-term. The final output of this step is a raw candidate transaction aggregated with previous transactions and given as input to the classification phase.

5.2 Oracles and Fraud Detectors.

In our system, we have two detectors. One is the *Bank Fraud Detector* used as a simulation of the system that the attacker is trying to bypass. The other is the *Oracle* that the attacker is using to build fraudulent transactions. Both detectors are based on some of the most used algorithms in literature [2] and deployed in banking institutions.

Random Forest. [7] model is an ensemble of decision trees. It combines the concept of bagging where individual models in an ensemble are built through sampling with replacement from the training data, and the random subspace method,

where each tree in an ensemble is built from a random subset of attributes. Thus, predictions are obtained by aggregating the outputs from individual trees in the ensemble. Majority voting is used to determine the prediction outcome (i.e., the label fraudulent or legitimate). This algorithm, from literature [31] seems to outperform the other in the field of fraud detection.

Neural Networks. They are learning models built of simple elements called neurons, which take as input a real value, multiply it by weight, and run it through a non-linear activation function. By constructing multiple layers of neurons, each of which receives part of the input variables, and then passes on its results to the next layers, the network can learn very complex functions. We used the sigmoid function as the activation function for the output layer in order to use this model for the classification.

Logistic Regression. It is a widely used technique in problems in which the dependent variable is binary. It computes the output using a logistic function. Based on a threshold, it possible to estimate probabilities and classify transactions.

XGBoost. EXtreme Gradient Boosting is based on an ensemble of methods. It trains and predicts using several models at once to produce a single better output. It exploits the concept of bagging and boosting to perform the classification. It has achieved excellent results in many domains [13].

Active Learning. It is a variant of AI^2 [32] applied to the banking dataset. It is an active learning approach that combines an ensemble of unsupervised learning (i.e., Autoencoder) with supervised learning techniques (Random Forests). Combining the anomaly scores computed by the unsupervised models, it ranks transactions and presents them to the analyst for review; subsequently, the feedback collected is used to train a Random Forest.

Banksealer. [11] It represents one of the systems currently deployed in the banking institution we collaborated with. It characterizes the users of the online banking web application using a local, global, and temporal profile built during a training phase. Each type of profile extracts different statistical features from the transaction attributes, according to the type of model built. It works under semi-supervised settings and, once, the profiles are built, it processes new transactions and ranks them according to their predicted risk of fraud. The experiments with this system are particularly insightful, since, as we will see in Section 6.4, they demonstrate that current solutions are not ready to “smart” attackers.

The Oracle and the bank fraud detector are characterized by the machine learning algorithm, the dataset used for training the model, and the feature extraction method. In Table 3, we show the complete summary of the Oracle and bank fraud detector models. We assign to each of them an ID that, from now on, we use to refer to them. For the hyperparameters tuning of those models, we use the holdout method, in which we select the last month of transactions as the validation set and the other data for the training set. For the models O1-O2-O3-O4-B1, which are Random Forest models, we use, respectively, 200, 500, 200, 100, 100 estimators, and a max depth of 14, 8, 14, 14, 14. The Neural Network (B2) has two hidden layers with 32 and 16 units and a ReLU (Rectified Linear Unit) function as the activation function. Finally, there is the output layer with a Sigmoid activation function. The two hidden layers have l2 regularizers with $\lambda = 10^{-4}$, the loss function we use is the binary cross-entropy optimized with Adam optimizer. The XGB classifier (B3) has 200 estimators and a max depth of 20, with a learning rate of 0.1. The Logistic Regression classifier (B4) has an l2 regularization with parameter $C = \frac{1}{\lambda} = 10$. The active learning model (B5) is based on an AutoEncoder with a single encoding layer with a size of 25 units; it has a dropout regularization with dropout rate = 0.2. To validate each model, we use the commonly used metrics of accuracy, precision, recall, and f1-score (see Appendix C). To estimate the performances of the detectors under attack, in Table 4, we show the validation scores of each model. It is interesting to notice that the results obtained by Banksealer (B6), currently deployed in a real banking environment, are the lowest between the considered algorithms. This is because it was not possible to tune it like other algorithms, but we kept the parameters left by banking analysts, which tends to distrust from transactions coming from foreign countries only.

6 Experimental Evaluation

We evaluate our evasive approach against the state-of-the-art fraud detectors described in Section 5.2 and simulating an attacker with different degrees of knowledge that perform attacks by following different strategies.

Table 3: Overview of the Oracle (O1-O4) and detectors models (B1-B6). For the feature extraction strategies see Table 2

ID	DATASET	FEATURES EXTRACTION	ALGORITHMS
STRATEGY			
Oracle			
O1	2012-13	A	RANDOM FOREST
O2	2014-15	A	RANDOM FOREST
O3	2012-13	B	RANDOM FOREST
O4	2012-13	C	RANDOM FOREST
Detectors			
B1	2014-15	B	RANDOM FOREST
B2	2014-15	B	NEURAL NETWORK
B3	2014-15	B	XGBOOST
B4	2014-15	B	LOGISTIC REGRESSION
B5	2014-15	C	ACTIVE LEARNING
B6	2014-15	BANKSEALER [11]	

6.1 Attack Scenarios

As described in Section 5.1 the attacker does not have the complete control of all the features: the beneficiary IBAN and IBAN_CC are fixed (usually a money mule), the IP address is a national address from which the attacker makes the connection (possibly using a VPN), the Session ID is generated for each transaction. Therefore, the features that can be fully manipulated by the attacker are the amount and the timestamp. The timestamps are selected by using the algorithm described in the Section 5. Regarding the amount, we set up three different scenarios that represent the strategies that an attacker could use to choose the amount and the number of frauds to be committed. In this way, we can compare the results of the approach against different choices of the amount.

Scenario 1. In this first scenario, the attacker has the goal to execute 20 transactions per user of € 2,500, so the idea is to do many transactions with a medium-low amount.

Scenario 2. The attacker has the goal to execute 10 transactions per user of € 10,000, so few transactions with medium amounts are executed.

Scenario 3. The attacker aims to execute 5 transactions per

Table 4: Scores of fraud detectors on validation data

MODEL ID	ACCURACY	PRECISION	RECALL	F1-SCORE
B1	99.7%	70.9%	96.2%	81.7%
B2	99.5%	63.9%	86.7%	73.6%
B3	99.6%	69.0%	93.6%	79.5%
B4	98.9%	34.1%	46.2%	39.2%
B5	99.5%	66.1%	89.7%	76.1%
B6	98.4%	8.5%	11.5%	9.8%

Table 5: Summary of all the scenarios

	SCENARIO 1	SCENARIO 2	SCENARIO 3
Total Victims	80	160	320
Total Frauds	1600	1600	1600
Frauds per User	20	10	5
Money per Fraud	€2,500	€10,000	€15,000
Money per Victim	€50,000	€100,000	€75,000

user of €15,000. The attacker wants to steal as much money as possible in the short term and tries to execute a few transactions with a high amount.

For each scenario, we inject about 1% (1600) of frauds³. We choose the parameters of the three scenarios to maintain the same overall number of frauds injected and the same total amount stolen. A summary of the configuration of the three scenarios is shown in Table 5.

6.2 Metrics

We evaluate the performance using 4 metrics: *Injection Rate*, *Evasion Rate*, *Attack Detection Rate*, and *Money Stolen*.

Injection Rate. It indicates the percentage of frauds that the attacker carries out against the user in relation to the number of frauds targeted - i.e., it is the proportion of frauds that the Oracle classifies as legitimate with respect to the number of frauds that the attacker wants to perform. This metric depends on the threshold that we set in the Oracle. The Oracle decides whether or not a fraud is likely to be uncovered. This metric is also useful to compare experiments based on the level of confidence. The lower the classification threshold we set, the lower the injection rate we have. In this way, the risk of fraud being detected drops within specified limits.

Evasion rate. It is the percentage of frauds concealed from the fraud detection system with respect to the frauds carried out against the user.

Attack Detection Rate. If we consider an attack as the set of frauds that the attacker performs against the single user, the *Attack Detection Rate* is the percentage of attacks that are detected by the fraud detection system. Therefore, the *Attack Detection Rate* can be seen as the probability that the system detects the attacker if he or she performs the attack against one single user.

Money Stolen. It represents the money in euro (€) that the bank loses. This metric is significant because it is dependent on all three previous metrics and the attack strategy of the attackers (Scenario). Also, it gives an idea of the monetary impact that these attacks can have on a real bank. With

³The 1% is chosen because it is a reasonable number of transactions that the bank can inspect manually with its specialized bank analysts.

these metrics, we can capture all the main aspects to compare the different experiments and measure the validity of our approach.

Defined with N the number of the targeted victims, F the number of frauds that the attacker wants to perform, K the amount of money for each fraudulent transaction, X_n the number of transactions classified as legitimate by the Oracle, and Y_n with $Y_n \leq X_n$ the number of transactions classified as legitimate by the fraud detector, we can define the metrics as follows:

$$Injection\ rate = 1/N \cdot \sum_n X_n / F$$

$$Evasion\ rate = 1/N \cdot \sum_n Y_n / X_n$$

$$Attack\ Detection\ Rate = \sum_n (X_n - Y_n) / N$$

$$Money\ Stolen = \sum_n Y_n \cdot K$$

6.3 Experimental Settings

We perform an attack for each scenario and each degree of knowledge of the attacker. The degree of knowledge of the attacker are described in Section 3; the scenarios are summarized in Table 5. The combinations of Oracles and detectors per degree of knowledge are summarized in Table 6. We inject a number of frauds approximately equal to 1% of transactions in the dataset, and we use different fraud detectors. In order to choose classification thresholds, we consider that the bank usually has the workforce to inspect about 1% of transactions that are carried out each month. Thus, after sorting transactions by anomaly score, we classify the first 1% of transactions as fraud and the remaining ones as legitimate. We train the fraud detectors the dataset 2014-15 using months from October to January, while February is the one subject to the evasion attacks. We randomly select the victims, excluding those users with less than 5 transactions in the dataset. We perform each experiment 10 times and compute the average value for each metric in order to have a more reliable estimate and not biased by the selected users.

Black-Box. The attacker has zero knowledge of the fraud detector, but he or she has obtained an old bank dataset (2012-13), different with respect the one used by the fraud detector. After having retrieved the transactions performed by the victim in January, he or she uses the model O1 as Oracle to perform the evasion attack.

Black-Box with Data. The attacker has no knowledge about the fraud detector and has retrieved the dataset (2014-15) used for the training both the Oracle and the fraud detector. Therefore, the attacker uses all transactions of the victim to train the Oracle, which in this case coincides with model O2.

Gray-Box. The attacker has acquired partial knowledge about the system. He or she knows how to extract the final features that the detector uses as input for the machine learning algorithm. Also, the attacker has retrieved the transactions performed by the victim in January. Then, he or she extracts the same features used by the fraud detector; the attacker uses

Table 6: Oracle used against each bank fraud detector depending on attacker’s knowledge

ORACLE				DETECTOR			
ID	DATASET	FEAT.	MODEL	ID	DATASET	FEAT.	MODEL
BLACK-BOX							
O1	2012-13	A	RF	B1	2014-15	B	RF
				B2	2014-15	B	NN
				B3	2014-15	B	LR
				B4	2014-15	B	XGB
				B5	2014-15	C	AL
				B6	2014-15	BANKSEALER	
BLACK-BOX WITH DATA							
O2	2014-15	A	RF	B1	2014-15	B	RF
				B2	2014-15	B	NN
				B3	2014-15	B	LR
				B4	2014-15	B	XGB
				B5	2014-15	C	AL
				B6	2014-15	BANKSEALER	
GRAY-BOX							
O3	2012-13	B	RF	B1	2014-15	B	RF
				B2	2014-15	B	NN
				B3	2014-15	B	LR
				B4	2014-15	B	XGB
O4	2012-13	C	RF	B5	2014-15	C	AL
WHITE-BOX							
B1	2014-15	B	RF	B1	2014-15	B	RF
B2	2014-15	B	NN	B2	2014-15	B	NN
B3	2014-15	B	LR	B3	2014-15	B	LR
B4	2014-15	B	XGB	B4	2014-15	B	XGB
B5	2014-15	C	AL	B5	2014-15	C	AL
B6	2014-15	BANKSEALER		B6	2014-15	BANKSEALER	

an Oracle based on the Random Forest algorithm, which has shown to be the best fraud detection system. We used model O3 and O4 as Oracles for attacking respectively fraud detectors B1-B2-B3-B4 and B5. We do not perform the Gray-Box experiment on the fraud detectors B6, since the partial knowledge acquired by the attacker can not be directly applied to perform the attack. In fact, the feature used by B6 can not be directly used by the Oracle without re-engineering them. This is due to the fact that B6 uses a combination of statistical and machine-learning-based detectors with ad-hoc features. Therefore, an attacker would resort to the Black-Box attack.

White-Box. The attacker has full knowledge of the system, including the dataset used for the training. The attacker reproduces the real system and uses it as Oracle so that he or she can perform a “perfect” attack.

6.4 Discussion on Results

In Table 7, we present the results of the experimental evaluation against the attacks for each scenario and degree of knowledge of the attacker (i.e., Black-Box, Black-Box with

data, Gray-Box, and White-Box).

Black-Box. Regarding the Black-Box attack, the best strategy for the attacker is the one represented in Scenario 3. In fact, except for the case in which the fraud detector is based on Logistic Regression (B4) and BankSealer (B6), in Scenario 3 we get attack detection rate values lower than in the other two Scenarios and much higher values in terms of money stolen. The values of evasion rate, however, remain stable in all three scenarios, except in the case where the attack is directed to the hybrid fraud detector (B5). This detector achieves excellent results in terms of evasion rate and attack detection rate in Scenario 1, where the attack is always detected (Attack Detection Rate = 100%). The performances of this fraud detector decrease significantly with the increase of the amount of the fraud; in Scenario 3, the Attack Detection Rate is reduced to 36%. The worst fraud detector is B4, which in Scenario 1 is completely evaded. The performance of the attack against it remains almost perfect also in the other two scenarios. Models B2 and B3 obtain similar values in all the metrics for all the scenarios.

Black-Box with Data. Even for this type of attack, the best strategy for the attacker is the one applied in Scenario 3, in which the money stolen reaches the highest values. The fraud detectors that best counteract the attack are B1 and B5. In Scenario 1, they have an evasion rate of 45% and 57% respectively, while the other detectors have values much higher 76%-89%-99%-99%-94%. Also, in this setting, the fraud detector B4 is easily evaded in all three scenarios, reaching at most an attack detection rate of 4% in Scenario 3. Models B2-B3 continue to be similar, obtaining almost the same values in all the scenarios.

Gray-Box. Also, for the Gray-Box attack, the best strategy for the attacker is the one applied in Scenario 3, which obtains the highest values in terms of money stolen. The worst performances of the attack are obtained in Scenario 2, against fraud detector B1, which has an attack detection rate of 93% and an evasion rate of 52%. We note significant deterioration in detection performances of detector B5, which has an evasion rate of 90% and a detection rate of 28%, values much lower than those obtained in other settings. Detector B4 is easily evaded, with an evasion rate of even 100% in all three scenarios. As in the other settings, detectors B2-B3 are evaded with an average evasion rate of 85%.

White-Box. Having complete knowledge of fraud detectors, we can perform perfect attacks. We can effectively test the robustness of each fraud detector against a perfect attacker, which steals the maximum amount of money but is never detected. By conservatively tuning the Injection Rate, we were able to achieve 100% Evasion Rate and 0% Attack Detection Rate against all detectors in all scenarios. The worst performances are obtained by model B4, which in Scenario 3 leads the bank to a loss of about 23 million euros. Much better are the performances of detector B6 that, instead, in Scenario 2

Table 7: Experimental results for all the evasion scenarios, attacker's knowledge, and fraud detectors: In **green** the best results from the detector point of view, in **red** the worst result from the detector point of view

		RANDOM FOREST (B1)	NEURAL NETWORK (B2)	XGBOOST (B3)	LOGISTIC REGRESSION (B4)	AL(B5)	BS [11](B6)
BLACK-BOX							
SCENARIO 1	Injection Rate	58.5%	58.5%	58.5%	58.5%	58.5%	58.5%
	Evasion Rate	63%	85%	80%	100%	54%	95%
	Attack Detection Rate	93%	44%	69%	0%	100%	6%
	Money Stolen	€ 650,507	€ 1,585,267	€ 1,265,751	€ 2,329,885	€ 339,544	€ 2,144,541
SCENARIO 2	Injection Rate	49.6%	49.6%	49.6%	49.6%	49.6%	50%
	Evasion Rate	60%	89%	84%	99%	71%	81%
	Attack Detection Rate	84%	27%	49%	1%	83%	20%
	Money Stolen	€ 2,722,586	€ 6,356,793	€ 5,107,460	€ 7,868,793	€ 2,330,346	€ 5,717,647
SCENARIO 3	Injection Rate	69.7%	69.7%	69.7%	69.7%	69.7%	71%
	Evasion Rate	62%	88%	87%	98%	84%	75%
	Attack Detection Rate	68%	22%	26%	5%	36%	26%
	Money Stolen	€ 8,081,496	€ 13,954,723	€ 13,379,473	€ 16,193,353	€ 11,774,757	€ 11,672,400
BLACK-BOX WITH DATA							
SCENARIO 1	Injection Rate	59.6%	59.6%	59.6%	59.6%	59.6%	59.5%
	Evasion Rate	57%	89%	76%	99%	45%	94%
	Attack Detection Rate	95%	38%	74%	1%	97%	7%
	Money Stolen	€ 540,849	€ 1,764,979	€ 1,108,691	€ 2,357,735	€ 408,383	€ 2,145,080
SCENARIO 2	Injection Rate	55.7%	55.7%	55.7%	55.7%	55.7%	56%
	Evasion Rate	57%	88%	78%	100%	65%	82%
	Attack Detection Rate	83%	33%	59%	1%	77%	20%
	Money Stolen	€ 3,330,573	€ 7,079,843	€ 5,391,816	€ 8,852,717	€ 3,132,262	€ 6,694,115
SCENARIO 3	Injection Rate	59.7%	59.7%	59.7%	59.7%	59.7%	60%
	Evasion Rate	59%	87%	83%	98%	78%	75%
	Attack Detection Rate	66%	24%	34%	4%	45%	30%
	Money Stolen	€ 6,329,751	€ 11,646,219	€ 10,498,652	€ 13,775,411	€ 8,651,346	€ 9,504,000
GREY-BOX							
SCENARIO 1	Injection Rate	56.8%	56.8%	56.8%	56.8%	41.7%	-
	Evasion Rate	65%	88%	84%	100%	64%	-
	Attack Detection Rate	92%	40%	63%	1%	91%	-
	Money Stolen	€ 697,747	€ 1,687,563	€ 1,413,702	€ 2,261,085	€ 500,553	-
SCENARIO 2	Injection Rate	60.5%	60.5%	60.5%	60.5%	42.2%	-
	Evasion Rate	52%	85%	73%	100%	78%	-
	Attack Detection Rate	93%	40%	70%	0%	49%	-
	Money Stolen	€ 2,778,662	€ 7,370,974	€ 5,427,974	€ 9,652,461	€ 3,739,336	-
SCENARIO 3	Injection Rate	68.6%	68.6%	68.6%	68.6%	63.6%	-
	Evasion Rate	65%	93%	93%	100%	90%	-
	Attack Detection Rate	66%	17%	20%	0%	28%	-
	Money Stolen	€ 8,268,764	€ 14,634,344	€ 14,062,925	€ 16,412,751	€ 11,777,417	-
WHITE-BOX							
SCENARIO 1	Injection Rate	39.3%	68.3%	59.4%	99.5%	30.2%	97.3%
	Evasion Rate	100%	100%	100%	100%	100%	100%
	Attack Detection Rate	0%	0%	0%	0%	0%	0%
	Money Stolen	€ 1,575,986	€ 2,534,932	€ 2,378,945	€ 3,986,408	€ 1,198,900	€ 3,892,000
SCENARIO 2	Injection Rate	31.2%	69.1%	57.0%	99%	31.7%	22%
	Evasion Rate	100%	100%	100%	100%	100%	100%
	Attack Detection Rate	0%	0%	0%	0%	0%	0%
	Money Stolen	€ 4,978,737	€ 10,923,554	€ 9,126,239	€ 15,830,758	€ 5,059,850	€ 3,520,000
SCENARIO 3	Injection Rate	32.2%	78.2%	73.2%	96.7%	27.8%	22%
	Evasion Rate	100%	100%	100%	100%	100%	100%
	Attack Detection Rate	0%	0%	0%	0%	0%	0%
	Money Stolen	€ 7,697,218	€ 18,237,258	€ 17,567,217	€ 23,195,258	€ 6,647,723	€ 5,280,000

and 3, significantly reduces the losses of the bank. The results show that the increase in the degree of knowledge of the attacker does not lead to significant improvements in attack performance. For example, in the attacks against model B2 based on Neural Network, we have an evasion rate of 85-89-88% for the Black-Box setting in Scenarios 1, 2, and 3, respectively and 88-85-93% in the Gray-Box one. The injection rate and the evasion rate are dependent on the classification threshold used by the Oracle to determine if a transaction is a fraud or legitimate and, therefore, to determine if the attacker has to carry out the transaction. The stolen money depends both on the injection rate and on the evasion rate and therefore gives us a basis for comparing the different attacks. Even looking at the stolen money, we do not notice excessive improvements in the attacks in which the attacker has a higher degree of knowledge of the system. So, we can state that the isolated knowledge of the dataset and the knowledge of the feature extraction method does not bring significant advantages in conducting an attack following our approach. However, if the attacker has both pieces of knowledge and also knows the machine learning algorithm employed by the fraud detector, he can perform a White-Box attack that instead manages to hide perfectly the frauds. The experiments also show that the bank would lose a significant amount of money if many fraudsters were using this method, but we must take into account that this attack is not easily deployable in the real world because there are significant obstacles. First of all, the attacker needs a banking dataset, which is very difficult to obtain because the banks keep their data very carefully and do not release them. Besides, this approach is perilous for the attacker, in the case of Black-Box the model based on random forest (B1) has an attack detection rate higher than 68%, this means that an attacker has about 32% chance of performing an entire attack without being detected and then prosecuted by law for the crime committed. One last important consideration concerns fraud detectors: the results show that if the attack is performed on a simple fraud detector, with low performance (e.g., Logistic Regression, B4) also the Black-Box attack gets excellent results: with an attack detection rate of 0% in the case of the first scenario, 1% in the second and 5% in the case of the third scenario. A similar consideration can be made on the system currently deployed by the banking institution we collaborated with (i.e., Banksealer, B6). Very different is the case of the fraud detector based on Random Forest (B1), which is much more robust detecting 93% of the attacks in the first scenario, 84% in the second and 68% in the third. So we can state that the choice of the fraud detector is crucial for a bank to reduce money losses. In our experiments, the fraud detector based on Random Forest has a bound of money loss (determined by White-Box tests) of about one-third of the one we have with the fraud detector based on Logistic Regression.

7 Limitations and Future Works

Besides the assumptions made in Section 5, there are few more things that need to be clarified. Even if we had two real datasets provided by a banking group, we had to rely on domain experts (bank operators) to enrich our datasets with synthetic frauds and compensate for the lack of labels. This represents only a partial limitation; although we did not have real fraud flagged, we accurately modeled synthetic frauds, and the standard behavior of users was real and legit. A possible limitation regards the source of datasets. Both datasets used in the experiments belong to the same bank but in different periods. Therefore, we were able to evaluate only the transferability of an attack in the same domain. Also, this limited data source partially affects the representativeness of the experiment and may underestimate real-world fraud detection mechanisms' effectiveness. Financial institutions have significantly more flexibility with training data to build effective models and re-train them periodically. Interesting future works can use heterogeneous datasets that span over a longer time frame, perhaps from two or more banks, to compare its performance results with the one presented in this work, also evaluating the impact of the training dataset on the effectiveness of the approach.

Unlike the current approach that models the spending behavior of each user, future works may investigate the possibility for an adversary to cluster banking customers to generate a generic model per cluster. This could generalize the results better and theoretically reduce some of the costs associated with the attacks. Also, an extension of the present work, which was focused on study how well the detector performed when being attacked by evasion attacks, may consist in an evaluation of the impact of the false positives of the different fraud detection systems employed by banks. Even if some detection systems are less susceptible to evasion attacks, they may be characterized by high false-positive rates, thus costing money in terms of analysis time and might be less likely adopted by financial institutions. Finally, we believe that a promising future work may explore how the behavior of users can change over time: in the long term, the change can derive from variation in the purchasing power of the user, in the short term there may instead be extraordinary expenses such as the purchase of a car. This phenomenon is called "concept drift" and must be taken into account by the fraud detection system. To manage concept drift, often, the bank adopts an online-training technique. The model is re-trained with the last transactions after ensuring that these are not frauds. An attacker could then exploit the Oracle approach to perform a data poisoning attack. He or she can conceal frauds and consequently shift in his favor the classification boundary of the fraud detector, creating the opportunities for new frauds. So it would be interesting to deepen the study of a data poisoning attack based on our Oracle approach.

8 Conclusions

In this paper, we developed a novel approach to perform evasion attacks by overcoming the issues we found in the application of AML techniques to the field of banking fraud detection. We validate this approach by simulating an attacker that performs the attack against state-of-the-art fraud detection systems under different conditions. Our approach assumes that the attacker has a banking dataset at his disposal and can control the transactions of his victims. The results of the experiments show that a reasonable evasion rate is reachable even in the case of a Black-Box attack, in which the attacker does not have any information on the target fraud detection system. These results are strictly dependent on the fraud detector and range from the 60% of evasion rate in the case of a fraud detector based on Random Forest to the 100% in the case of a fraud detector based on Logistic Regression. A daunting fact for a real attacker is that the probability that the attack is detected after a certain number of successful frauds is about 66% in the case of fraud detectors based on Random Forest, so it is very inconvenient to use the approach to execute a real attack. An interesting future challenge would be the study of a data poisoning attack based on our approach. Since many fraud detectors use online training (i.e., they retrain regularly using the transactions that have been classified as legitimate), it may be possible to apply our approach to conceal frauds and study how to drift the classification threshold of the detector in order to compromise the detection performance.

References

- [1] Alejandro Correa Bahnsen, Djamila Aouada, Aleksandar Stojanovic, and Bjorn Ottersten. Feature engineering strategies for credit card fraud detection. *Expert Systems with Applications*, 51:134–142, 2016.
- [2] Siddhartha Bhattacharyya, Sanjeev Jha, Kurian Tharakunnel, and J Christopher Westland. Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3):602–613, 2011.
- [3] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrđić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.
- [4] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4):984–996, 2013.
- [5] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.
- [6] R Brause, T Langsdorf, and Michael Hepp. Neural data mining for credit card fraud detection. In *Proceedings 11th International Conference on Tools with Artificial Intelligence*, pages 103–106. IEEE, 1999.
- [7] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
- [8] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.
- [9] Michele Carminati, Alessandro Baggio, Federico Maggi, Umberto Spagnolini, and Stefano Zanero. Fraudbuster: temporal analysis and detection of advanced financial frauds. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 211–233. Springer, 2018.
- [10] Michele Carminati, Roberto Caron, Federico Maggi, Ilenia Epifani, and Stefano Zanero. Banksealer: An online banking fraud analysis and decision support system. In *IFIP International Information Security Conference*, pages 380–394. Springer, Berlin, Heidelberg, 2014.
- [11] Michele Carminati, Roberto Caron, Federico Maggi, Ilenia Epifani, and Stefano Zanero. Banksealer: A decision support system for online banking fraud analysis and investigation. *computers & security*, 53:175–186, 2015.
- [12] Michele Carminati, Mario Polino, Andrea Continella, Andrea Lanzi, Federico Maggi, and Stefano Zanero. Security evaluation of a banking fraud analysis system. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):11, 2018.
- [13] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.
- [14] Andrea Continella, Michele Carminati, Mario Polino, Andrea Lanzi, Stefano Zanero, and Federico Maggi. Prometheus: Analyzing webinject-based information stealers. *Journal of Computer Security*, 25(2):117–137, 2017.
- [15] Andrea Dal Pozzolo, Olivier Caelen, Yann-Ael Le Borgne, Serge Waterschoot, and Gianluca Bontempi. Learned lessons in credit card fraud detection from a practitioner perspective. *Expert systems with applications*, 41(10):4915–4928, 2014.
- [16] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

- [17] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 321–338, 2019.
- [18] Jose R Dorronsoro, Francisco Ginel, C Sgnchez, and Carlos S Cruz. Neural fraud detection in credit card operations. *IEEE transactions on neural networks*, 8(4):827–834, 1997.
- [19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- [20] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.
- [21] Ron Kohavi and George H John. The wrapper approach. In *Feature extraction, construction and selection*, pages 33–50. Springer, 1998.
- [22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [23] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.
- [24] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519. ACM, 2017.
- [25] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.
- [26] Kuldeep Randhawa, Chu Kiong Loo, Manjeevan Seera, Chee Peng Lim, and Asoke K Nandi. Credit card fraud detection using adaboost and majority voting. *IEEE access*, 6:14277–14284, 2018.
- [27] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.
- [28] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*, pages 305–316. IEEE, 2010.
- [29] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
- [30] Yaniv Taigman, Ming Yang, Marc’ Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.
- [31] Véronique Van Vlasselaer, Cristián Bravo, Olivier Caelen, Tina Eliassi-Rad, Leman Akoglu, Monique Snoeck, and Bart Baesens. Apate: A novel approach for automated credit card transaction fraud detection using network-based extensions. *Decision Support Systems*, 75:38–48, 2015.
- [32] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. Ai²: training a big data machine to defend. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*, pages 49–54. IEEE, 2016.
- [33] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in neural information processing systems*, pages 2773–2781, 2015.
- [34] David J Weston, David J Hand, Niall M Adams, Christopher Whitrow, and Piotr Juszczak. Plastic card fraud detection using peer group analysis. *Advances in Data Analysis and Classification*, 2(1):45–62, 2008.
- [35] Christopher Whitrow, David J Hand, Piotr Juszczak, D Weston, and Niall M Adams. Transaction aggregation as a strategy for credit card fraud detection. *Data mining and knowledge discovery*, 18(1):30–55, 2009.
- [36] Vladimir Zaslavsky and Anna Strizhak. Credit card fraud detection using self-organizing maps. *Information and Security*, 18:48, 2006.

A Features extracted

The **directly** derivable features we use are:

- **amount.** Amount of the transaction in euro (€), without any type of transformation.
- **time_x, time_y.** These two features are computed from the Timestamp. The time is cyclical and we must be careful when deciding how to encode it as input to the machine learning model. If we encode hours as integers we have a problem: the distance computed, for example between 23 and 22 is $23 - 22 = 1$, while the distance that it calculates between midnight and 23 is $0 - 23 = -23$. We need to change the encoding of the features such that midnight and 23 are the same distance apart as any other two hours. A common method for encoding cyclical data is to transform the data into two dimensions using a sine and cosine transformation. So $time_x$ and $time_y$ are: $t \leftarrow ts_h \cdot 3600 + ts_{min} \cdot 60 + ts_{sec}$, $time_x = \cos \frac{t-2\pi}{86400}$, $time_y = \sin \frac{t-2\pi}{86400}$
- **is_national_iban.** A Boolean value indicating if the beneficiary's IBAN is national.
- **is_national_asn.** A Boolean value indicating if the connection of the coming transaction has a national IP Address.
- **is_international.** A Boolean value indicating if the connection comes from a country different from the one of the beneficiary IBAN.
- **operation_type.** It is one-hot-encoded, so the three operations become respectively op_type_1 , op_type_2 , op_type_3
- **confirm_sms.** A Boolean value indicating if the transaction requires a confirmation sms.

The features obtained by **aggregating** transactions are:

- **group_function_time.** Consider *function*, *group*, *time* as variable. A *function* is computed on the user transactions grouped by *group* for a rolling window of size *time*.
- **group** can be: IBAN, IBAN_CC, IP, CC_ASN, SessionID or *none*, if we want to group only by user and take all the transactions of that user.
- **function** can be:
 - count. It counts the number of transactions in the considered window
 - sum. It computes the sum of the transactions amounts in the considered window
 - mean. It computes the mean of the transactions amounts in the considered window
 - std. It computes the standard deviation of the transactions amounts in the considered window

- **time** can be: 1 hour, 1 day, 7 days, 30 days and *none*, if we want to aggregate all the transaction regardless of the time
- **time_since_last_group.** Indicates the time in hours elapsed from the last transaction among those obtained by grouping the transactions by user and by *group*.
- **group_distance_from_mean.** Indicates the L1 distance of the transaction amount with respect to the mean amount of the transactions obtained by grouping the transactions first by user then by *group*.
- **is_new_iban.** A Boolean value indicating if it is the first time the user made a transaction to that IBAN.
- **is_new_iban_cc.** A Boolean value indicating if it is the first time the user made a transaction to an IBAN from that country.
- **is_new_ip.** A Boolean value indicating if it is the first time the user made a transaction coming from that IP Address.
- **is_new_cc_asn.** A Boolean value indicating if it is the first time the user made a transaction coming from that country.

B Adversarial Machine Learning Approaches

There are many studies on how to craft adversarial samples.

Szegedy *et al.* [29] generate adversarial examples using box-constrained optimization method Limited Memory Broyden Fletcher Goldfarb Shanno (LBFGS). Given an image x and a function $f(x)$, that maps image pixels vector to a discrete label, and assuming that f has an associated loss function denoted by $loss_f$, their method finds a different image x' similar to x under L2 distance, yet is labeled differently by the classifier. They model the problem as a constrained minimization problem: minimize $\|x - x'\|_2^2$, subject to $f(x') = l$ and $x' \in [0, 1]^n$. In particular, they found an approximate solution by solving the analogous problem: minimize $\|x - x'\|_2^2 + loss_{F,l}(x')$, subject to $x' \in [0, 1]^n$. The final result is the minimum $c > 0$ for which $x - x'$ satisfies $f(x') = l$, where l is the target label.

Goodfellow *et al.*, proposed the Fast Gradient Sign Method (FGSM) [19] that is designed primarily to be as fast as possible instead of accurately producing an adversarial sample (i.e., it is not meant to produce minimal perturbed adversarial samples). Moreover, it is optimized for the L_∞ distance metric. Given an image x , it computes the adversarial sample $x' = x - \epsilon \cdot \text{sign}(\nabla \text{loss}_{F,l}(x))$, where t represents the target

Table 8: Success Rate and Transferability of the three AML methods

	SUCCESS RATE	TRANSFERABILITY
GRADIENT METHOD	99.3%	16.1%
JSMA	98.2%	16.1%
CARLINI & WAGNER L2	12%	9.3%

label and ϵ is chosen to ensure misclassification without excessively altering the image.

The attack proposed by Papernot *et al.* [25] is known under the name of Jacobian Saliency Map Approach (JSMA) and it is based on a L_0 distance. This attack is based on a greedy algorithm in which, at each iteration, a saliency map is built based on the gradient of the output neural network with respect to the input image. This saliency map indicates the impact of each pixel (i.e., how much each pixel influences the outcome of the classification) and, at each iteration, the most significant pixel is perturbed. The method iterates until either the sample is misclassified by the classifier or pixels are modified more than a threshold.

Carlini&Wagner [8] designed three attacks tailored to three distance metrics: L_2, L_0, L_∞ distances. In L_2 Attack, given an image sample x , a target class t s.t. $t \neq C^*(x)$, they find w by solving this optimization problem: minimize $\|\frac{1}{2}(\tanh w + 1) - x\|_2^2 + c \cdot f(\frac{1}{2}(\tanh w + 1))$, where f , which is defined as $f(x') = \max(\max\{Z(x')_i : i \neq t\} - Z(x')_t, -\kappa)$, is the objective function and κ is a parameter that allows to tune the confidence of the adversarial sample. In L_0 Attack, since the L_0 distance metric is non-differentiable it is not possible to use the standard gradient descent, so they decided to use an iterative algorithm. At each iteration they identify, through the L_2 attack, which pixel influence less the output of the classifier and fix its value so that it will never be changed. By doing this, the set of fixed pixels grows at each iteration, until a minimal subset of modifiable pixels is identified. In L_∞ Attack, since the L_∞ distance metric is not fully differentiable and standard gradient descent does not perform, a different iterative attack is put in place. At each iteration the following optimization problem is solved: minimize $c \cdot f(x + \delta) + \sum_i [(\delta_i - \tau)^+]$, in which if $\delta_i < \tau$ for all i , they reduce τ by a factor of 0.9 and repeat; otherwise they terminate the search.

B.1 Evaluation of standard AML approaches for Fraud Detection

As highlighted in Section 2, the issues in performing evasion attacks based on standard AML algorithms in the fraud detection domain are mainly practical. It would be challenging (if not infeasible) for an attacker to perform a real bank transfer (i.e., a real transaction) representing the adversarial sample returned as output by the AML algorithm. The

Algorithm 2 Craft Adversarial Samples. X is the benign sample, Y^* is the NN output, F is the function learned by the NN during training, Υ is the maximum distortion, θ is the change made to features, C is a confidence parameter

```

1: procedure GRADIENTMETHOD( $X, Y^*, F, \Upsilon, \theta, C$ )
2:    $X^* \leftarrow X$ 
3:   while  $F(X^*) \neq Y^* + C$  and  $\|\delta_X\| < \Upsilon$  do
4:      $G = \nabla F(X^*)$ 
5:      $S = \text{sign}(G)$ 
6:      $A = \text{abs}(S)$ 
7:      $X^*_{*i} \leftarrow X^*_{*i} + \theta_{i \text{ s.t. } i_{\max} = \text{argmax}_i(A)} \cdot S$ 
8:      $\delta_X \leftarrow X^* - X$ 
9:   end while
10:  return  $X^*$ 
11: end procedure

```

reason resides in which features are modified and how. As shown in Figure 3, the perturbed features are the aggregated ones; hence, an attacker, to perform an evasion attack, should then extract the direct features that correspond to valid, real transactions under the constraints that, once aggregated, they produce coherent aggregated features. If we add the practical domain constraints presented in Section 2, this problem may be not feasible in terms of resources and return for an attacker. However, for the sake of completeness, we evaluate AML techniques to measure the theoretical performance of such attacks in our domain. In particular, we make use of the following algorithms: Carlini & Wagner L2 [8], Jacobian Saliency Map Approach [25], and a simple approach based on the computation of the gradient whose algorithm is shown by Pseudocode 2.

For this evaluation, we use two models: a Neural Network, necessary for applying the AML algorithms, and a Random Forest classifier that acts as the bank fraud detector. We used the dataset 2012-13 as the training set and the same features used for the Oracle in the previous experiments, shown in Table 2, to train the Artificial Neural Network with two hidden layers: the first with 32 units and the second with 16 units. We used ReLU as the activation function for the hidden layers, while for the output layer, we used the Softmax function and two neurons. We extracted frauds from dataset 2012-13, then we perturbed these frauds using the three AML approaches that are based on the neural network to compute the gradients. In order to test transferability, we used, as a bank fraud detector, the model O1 from Table 3, which is trained on the same dataset as the Neural Network and also uses the same aggregation method. In order to evaluate our results, we used the metrics defined by Papernot *et al.* [24]: *success rate* and *transferability*. The *success rate* is the proportion of adversarial samples misclassified by the Neural Network. The *transferability* of adversarial samples refers to the random forest (O1) misclassification rate of adversarial samples crafted using the Neural Network. The results in Table 8 show

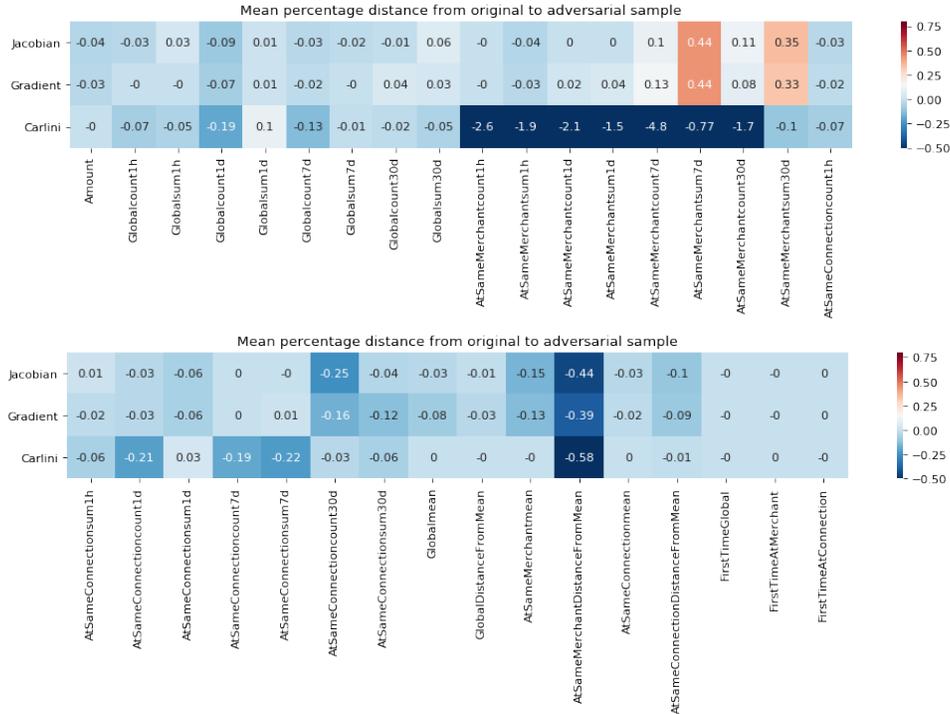


Figure 3: The heatmap shows how much each feature has been perturbed as a percentage of its initial value. For example, 1.0 means that the value of that feature has been doubled, it is increased by 100% compared with the initial value

poor performance, and also, to achieve those results, we need to apply wide perturbations to the initial samples, as we can see from the heatmap in Figure 3.

C Machine Learning Metrics

We present the most common metrics used to evaluate the quality of a machine learning model. The terms True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), are used to compare the results of the classifier under test with trusted ground truth. The terms *positive* and *negative* refer to the classifier prediction while the terms *true* and *false* refer to whether that prediction corresponds to the external ground truth. In our work, the positive class represents the frauds, and the negative represents legitimate transactions. The most used metrics derived with those terms are:

- **Accuracy.** It represents the percentage of correctly

classified instances and it is defined as $Accuracy = \frac{tp+tn}{tp+tn+fp+fn}$.

- **Precision.** It is also called positive predictive value and is defined as the fraction of relevant instances among the retrieved instances and it can be defined as: $Precision = \frac{tp}{tp+fp}$.
- **Recall.** It is also known as sensitivity or true positive rate. It is defined as the fraction of the total amount of relevant instances that were actually retrieved and can be defined as: $Recall = \frac{tp}{tp+fn}$.
- **F1-Score.** It is the harmonic mean of the precision and recall. F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0 and can be defined as: $F1 - Score = 2 \cdot \frac{precision \cdot recall}{precision+recall}$

The Limitations of Federated Learning in Sybil Settings

Clement Fung
Carnegie Mellon University
clementf@andrew.cmu.edu

Chris J.M. Yoon
University of British Columbia
yoon@alumni.ubc.ca

Ivan Beschastnikh
University of British Columbia
bestchai@cs.ubc.ca

Abstract

Federated learning over distributed multi-party data is an emerging paradigm that iteratively aggregates updates from a group of devices to train a globally shared model. Relying on a set of devices, however, opens up the door for sybil attacks: malicious devices may be controlled by a single adversary who directs these devices to attack the system.

We consider the susceptibility of federated learning to sybil attacks and propose a taxonomy of sybil objectives and strategies in this setting. We describe a new DoS attack that we term *training inflation* and present several ways to carry out this attack. We then evaluate recent distributed ML fault tolerance proposals and show that these are insufficient to mitigate several sybil-based attacks. Finally, we introduce a defense against targeted sybil-based poisoning called *FoolsGold*, which identifies sybils based on the diversity of client updates. We show that FoolsGold exceeds state of the art approaches when countering several types of poisoning attacks. Our work is open source and is available online: <https://github.com/DistributedML/FoolsGold>

1 Introduction

To train multi-party machine learning (ML) models from user-generated data, clients share their training data with services, which can be computationally expensive and privacy-violating. Federated learning (FL) [10, 39, 40] is a recent solution to both problems: data is kept on the client device and only model parameters are transferred to a central aggregator while training. Clients maintain a basic level of privacy by computing their model updates locally and independently, enabling collaborative ML in resource-constrained settings such as over a mobile network or in IoT(Internet of Things) deployments [24, 29, 49, 57].

However, FL widens the attack surface of the machine learning process: clients, who previously acted only as passive data providers, can now observe intermediate model states and adaptively contribute arbitrary information as part of de-

centralized training. For example, adversaries posing as honest clients can send erroneous updates to *poison* the trained model [3, 8, 15, 19, 23, 28, 36], *invert* or reconstruct the data of honest clients [26, 41, 42, 48, 54, 56], or *gain access to models* without usefully contributing [34].

Previous work has shown that adversaries who control more clients in a federated learning deployment can carry out poisoning attacks with more damage [52], and they can mount more elaborate and elusive attacks with coordinated malicious clients [58]. Such *sybil-based* attacks [18], in which an adversary controls multiple malicious clients, have been mentioned only in passing in existing work on FL [6, 30, 33, 35]. In this paper we focus on sybil attacks and contribute a taxonomy of sybil strategies that can be used to exacerbate known attacks against FL. As part of this process we define a new category of sybil-specific denial of service (DoS) attack that we term *training inflation*.

Having defined a space of sybil strategies and corresponding malicious objectives (e.g., model poisoning) in the context of FL, we consider the defenses. Existing work has explored several ways to defend the FL training process [9, 50, 60, 61]. We evaluate these defenses in the context of sybil-based attacks. Our results demonstrate that none of these defenses are effective, particularly against a large number of sybils.

Consequently, we propose a defense called **FoolsGold** to counter one class of sybil-based attacks on FL: *targeted* poisoning by sybil clones. In a targeted poisoning attack, these clones contribute updates towards a specific poisoning objective. In expectation over the training process, this targeting reveals sybils through behavior that is more similar to each other than the similarity observed amongst the honest clients. FoolsGold’s insight is to use this characteristic behavior to detect and reject poisoned contributions by adapting client learning rates based on *client contribution similarity*.

Our evaluation shows that FoolsGold defends FL from targeted poisoning by a large number of sybils, with only minimal change to the server-side algorithm and no change to the client-side algorithms. We evaluate FoolsGold on 4 diverse data sets (MNIST [31], VGGFace2 [14], KDDCup99 [17],

Amazon Reviews [17]) and 3 model types (1-layer Softmax, SqueezeNet, VGGNet) and find that our approach mitigates poisoning attacks under a variety of conditions, including different distributions of client data, varying poisoning targets, and various sybil strategies.

In summary, we make the following contributions:

- ★ We provide a taxonomy of sybil-based attacks on federated learning, which includes sybil strategies paired with an objective based on existing work on attacks against FL. We use this taxonomy to motivate open research problems in this space, and contribute a new type of DoS attack that we term *training inflation*.
- ★ We evaluate existing defenses against malicious sybil-based attacks on ML (Multi-Krum [9], median [62], trimmed mean [62]).
- ★ We design, implement, and evaluate a novel defense against sybil-based targeted poisoning attacks for federated learning that uses an adaptive learning rate per client based on inter-client contribution similarity.
- ★ In the context of colluding sybils, we design and evaluate intelligent poisoning attacks performed across sybils, and show that FoolsGold can defend against them, while suggesting strategies for further mitigation.

2 Background

Machine Learning (ML). Many ML problems are the minimization of a loss function in a large Euclidean space. For an ML classification task that predicts a discrete class; prediction errors result in a higher loss. Given a set of training data and a proposed model, ML algorithms *train*, or find an optimal set of parameters, for the given training set.

Stochastic gradient descent (SGD). One approach in ML is to use stochastic gradient descent (SGD) [12], an iterative algorithm that selects a batch of training examples, uses them to compute gradients on the parameters of the current model, and takes gradient steps in the direction that minimizes the loss function. The algorithm then updates the model parameters and another iteration is performed. SGD is a general learning algorithm that can be used to train a wide variety of models, including deep neural networks [12]. We assume SGD as the optimization algorithm in this paper. In SGD, the model parameters w are updated at each iteration t as follows:

$$w_{t+1} = w_t - \eta_t \left(\frac{1}{b} \sum_{(x_i, y_i) \in B_t} \nabla l(w_t, x_i, y_i) + \lambda \|w_t\|_p \right) \quad (1)$$

where η_t represents a local learning rate, λ is a regularization parameter on an L_p norm of the parameters that prevents over-fitting, B_t represents a gradient batch of training data examples (x_i, y_i) of size b and ∇l represents the gradient of the loss function.

Federated learning [39]. We assume a standard federated learning setting, in which training data is distributed across multiple clients and the aggregator does not see any training data.

The distributed learning process is performed by a set of clients over *synchronous update rounds*, in which an aggregation of the k client updates, weighted by their proportional dataset size $\frac{n_k}{n}$, is applied to the model atomically.

$$w_{g,t+1} = w_{g,t} + \sum_k \frac{n_k}{n} \Delta w_{k,t}$$

Even if the training data is distributed such that it is not independent and identically distributed (non-IID), federated learning can still attain convergence. For example, federated learning can train an MNIST [31] digit recognition classifier in a setting where each client only holds data of 1 of the digits (0-9).

Federated learning comes in two forms: FEDSGD, in which each client sends every SGD update to the server, and FEDAVG, in which clients locally aggregate multiple SGD iterations before sending updates to the server, which is more communication efficient [39].

3 Threat model for sybil-based attacks on FL

Setting assumptions. We are focused on FL and therefore assume that data is distributed across clients and hidden, such as in an IoT deployment with multiple devices distributed in people’s homes. The adversary can only access and influence the model state through the FL API. They cannot observe the training data of other honest clients. The adversary can observe the global change in model state to learn the total averaged update across all clients, but they cannot view individual honest client updates.

We assume that the server is uncompromised and is not malicious. In general, sybils can also be prevented non-algorithmically through techniques such as CAPTCHAs or device-specific asymmetric keys. We assume that such account or device verification services in the FL system do not exist, or that the adversary has the means to bypass these solutions.

Attacker capability. A system like FL, that allows clients to join and leave, is susceptible to sybil attacks [18] in which an adversary gains influence by joining a system using multiple colluding aliases. Sybil attacks are especially prevalent in IoT sensor networks [44], an emerging use case for FL [13, 33]. In this work, we assume that an adversary leverage sybils to mount more powerful attacks on FL.

While an attacker’s influence on the system may increase with more sybils, the information the sybils learn from the system remains constant: a snapshot of the global model. The attack strategies are therefore more limited. The same holds for defenses: since no auxiliary information is assumed, sybil

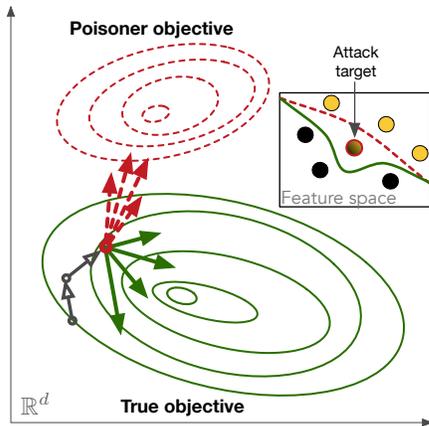


Figure 1: Targeted poisoning attack in SGD. The dotted red vectors are sybil contributions that drive the model towards the poisoner’s objective. The solid green vectors are contributed by honest clients that drive towards the true objective.

Target	Attack type	Attack objective
Model quality	Untargeted poisoning [19,59]	Decrease model accuracy
	Targeted poisoning [3,6,36,58,\$7.3,\$7.4]	Decrease model accuracy on target class only
Privacy	Data inversion [26,48,56]	Learn data from clients
	Membership inference [41,42,54]	Determine if a client has certain data
Utility	Model free riding [34]	Access model without usefully contributing
Resource	Training time inflation \$5.2	Increase training time
	Bandwidth inflation	Increase bandwidth used by server/clients
	CPU inflation	Increase CPU usage at server/clients

Table 1: Types of attacks against Federated Learning.

defenses based on e.g., information from social networks or peer-to-peer systems are not applicable [22, 53, 55, 63, 64].

Next, we review different adversarial objectives and examples of attack variants that achieve these objectives in the context of FL.

4 Attack objectives and strategies in FL

FL is susceptible to a variety of attack objectives and strategies. We now review the attack objectives in Table 1. In Section 4.2 we will review the strategies in Table 2 and discuss their intersection in Table 3.

Dimension	Sybil strategy	Description
Data	Clones	Sybils perform optimization steps on <i>identical local dataset</i>
	Act-alikes	Sybils perform optimization steps on <i>different local dataset</i>
	Clowns	Sybils use <i>synthetic gradients</i> not based on dataset
Coordination	Uncoordinated	Sybils act <i>without coordination</i>
	Swarm	Sybils coordinate and <i>weakly synchronize</i> states
	Puppets	Sybils coordinate and <i>synchronize on each round</i>
Churn	Remainers	Sybils join and stay
	Churners	Sybils join and leave

Table 2: High-level sybil strategies in Federated Learning.

4.1 Attack objectives

Attacking trained model quality. Adversaries may attack the quality of the model by supplying strategic inputs during training. In *untargeted poisoning*, the adversary aims to produce a model with low test accuracy that performs poorly across all classes [19, 59].

In *targeted poisoning*, the adversary directs the shared model towards a more specific objective. The adversary’s goal is to produce a model where a subset of inputs are classified as a different, incorrect class. For example, this subset could be a source class (label-flipping attack [5]) or a set of images with a curated pattern (backdoor attack [3, 23]). To avoid detection of such poisoning attacks, the prediction accuracy of classes unrelated to the attack should not change. In FL, each client has an equal share of the aggregated gradient and attackers can attack any class with enough influence by generating additional sybils as shown in Figure 1.

Later in the paper we will demonstrate the vulnerability of FL to targeted poisoning by sybils (Section 5.3) and design and evaluate a new defense, *FoolsGold*, to defend against these types of attacks (Sections 6 and 7).

Attacking privacy of honest clients. Clients in FL possess a subset of data that is not explicitly shared with other clients. This enables learning over private datasets [39]. Although adversaries cannot access the training data at honest clients, they may attack the model and infer sensitive client information from the changes in the model state. An adversary may even influence the shared model to have it leak more private information in future iterations.

In a *data inversion* attack, the adversary reconstructs the training data of a targeted honest client [27] by generating a sample that closely represents the training data of a specific client. Alternatively, in a *membership inference* attack, the adversary determines whether or not a client has a specific datapoint in their dataset [41].

Table 2 <i>Sybil strategies</i>			Table 1 <i>Attack types</i>					
Churn	Data	Coordination	U.Poison	T.Poison	D.Inversion	M.Infer	M.Free	T.Inflate
Remainers	Clones	Uncoordinated Swarm Puppets		FoolsGold \$6			[41]	
	Act-alikes	Uncoordinated Swarm Puppets		[58]				
	Clowns	Uncoordinated Swarm Puppets		[3,6,36]	[26,48,56]	[41,42,54]	[34]	\$5.2
Churners	All	All	[19,59]	\$7.3, \$7.4				\$5.2
					Unexplored			

Table 3: Known combinations of sybil strategies (Table 2) and attacks (Table 1). FoolsGold (Section 6) is highlighted as a contribution in this paper that defends against remainder clones whose goal is targeted poisoning. The other highlighted+bolded sections in this paper are new attack variants we contribute in this paper.

Attacking system utility. FL assumes that the utility of model contributions is equally distributed amongst participating clients: all clients contribute data of relative equal value to the shared model and receive a similar increase in utility when gaining access to the shared model. In a *model free-riding* attack, the adversary participates in the FL process while providing contributions of negligible value [34]. When the training process completes, the adversary gains access to the shared model, despite providing no value to the system.

Attacking resources in the system. Alleviating resource usage in FL is an active area of research. Recent work has introduced model compression techniques and dropout schemes to extend FL to resource constrained settings [13].

With this in mind, we propose a class of attacks against the FL process itself. The aim of these attacks is to inflate the *training time*, *used bandwidth*, or *used compute cycles*. These attack variants can be seen as a form of denial of service, since they waste resources intended for FL at the server and honest clients. While the utility and efficacy of these attacks depend on the configuration of the FL system [13], we show in Section 5.2 that for some standard heuristic of early stopping criteria, sybils can inflate the training time arbitrarily by influencing these heuristics.

4.2 Sybil attack strategies

There are many ways in which sybils may be used to implement the attacks from the previous section. In this section we provide a preliminary taxonomy of *how* an adversary may mount a sybil-based attack (Table 2). We also provide an intersection of these strategies and their known attack variants (from Table 1) in Table 3. We consider three dimensions that we believe are key to understanding sybil attacks (and defending against them): data distribution among sybils, level

of sybil coordination, and level of sybil churn.

Data distribution among sybils. In FL, clients, whether honest or malicious, provide *updates* to the shared model. These updates are assumed to be the result of an optimization algorithm based on client training data. They may be based on identical datasets at sybils (*clones*), different datasets (*act-alikes*), or may even be generated synthetically by sybils through an algorithm (*clowns*). Most known attacks on FL use synthesized gradients to achieve their attack objective [3, 19, 59].

Level of sybil coordination. An adversary that generates an increasing number of sybils, increases their influence on the system. This influence may be a type of a brute force strategy (*uncoordinated*), in which the honest clients are simply overpowered by the sybils on their desired objective. Alternatively, sybils may communicate amongst themselves before generating the next series of model updates. The state shared between sybils may be weakly consistent (*swarm*), through infrequent synchronization, or strongly consistent (*puppets*) through frequent synchronization, enabling more advanced attacks that require complex computation and communication.

The majority of attacks on FL use uncoordinated sybils. While some attacks use synchronized strategies [19, 59], these are limited to untargeted poisoning. In Sections 7.3 and 7.4, we demonstrate the power of these strategies by designing novel synchronized attacks against our FoolsGold defense.

Churn level of sybils. Although sybil churn is presently not used by defenses or attacks on FL, the churn level may impact defenses that rely on stateful client tracking during the learning process. Sybils that remain in the system (*remainers*) for an extended period of time may use their stability to build up their reputation with the server. Sybils that join, leave, and re-join the system (*churners*) may use churn to prevent the server from linking their activity across sessions.

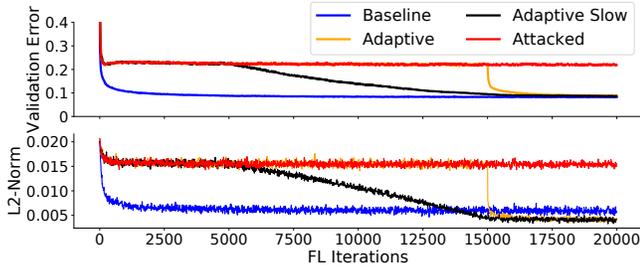


Figure 2: Training inflation attack on FL with 5 sybils.

5 Can current defenses handle sybils?

We show next that, even when only assuming attacks from uncoordinated clones, current defenses are inadequate in defending FL from sybils.

5.1 Existing defenses for FL

In the general ML setting, defenses rely on access to the training data [4] or access to the training process itself [7, 25, 45] to defend against adversaries. Since FL does not have access to these elements, FL defenses are limited to those that use *robust aggregation schemes*, as these operate on the server only. Several such aggregation techniques have been proposed to defend against Byzantine adversaries in FL, including Multi-Krum [9], median [62], and trimmed mean [62].

At each iteration of **Multi-Krum aggregation**, the total Euclidean distance from the $n - f - 2$ nearest neighbors is calculated for each client contribution. The f updates with the highest distances are removed and the average of the remaining updates is calculated.

When using the **median aggregation** technique, the element-wise median (the global update value for a parameter ΔW_j is the median of Δw_j) across all clients is used as the global update. Similarly, when using the **trimmed mean aggregation** technique, the highest and lowest β values for each feature are removed prior to computing the aggregated mean.

For all three techniques above, a successful defense requires an explicit bound on the maximum number of Byzantine clients. We show that if an adversary can spawn an arbitrarily large number of sybils, these techniques fail to work.

5.2 Training inflation attacks by sybil clones

Sybils can attack FL training by performing a *training inflation attack*. In this section, we show that when typical convergence heuristics are used, sybil clones can inflate the training time for as long as they wish, consuming shared resources on both the server and clients.

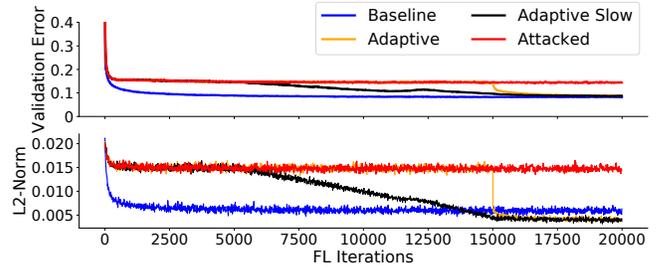


Figure 3: Training inflation attack with 5 sybils on FL with Multi-Krum when $f = 2$.

An ML training process needs a stopping condition, such as a fixed-length heuristic like the number of iterations or the number of training epochs. For better training efficiency, the process may also use a dynamic early stopping heuristic, typically based on the norm of the gradients [37], or the validation error [47].

We show both the average L2 norm of model updates and the validation error across iterations on both a baseline FL system (Figure 2) and an FL system with Multi-Krum (Figure 3). In evaluating Multi-Krum as a defense, when the parameter f is greater than the number of sybils, the attack is prevented. Figure 3 shows that Multi-Krum is ineffective for the case when an adversary can command more than f sybils.

In this experiment, 10 honest clients with a uniform sample of the MNIST dataset train a 1-layer softmax classifier; 5 sybils join the system and perform untargeted poisoning that causes the training to continue indefinitely (“Attacked”). Since this strategy is likely to be noticed by the server, we also show that sybils may choose to stop poisoning the model, either by sending negligibly small gradients (“Adaptive”) or by slowly reducing their own learning rate (“Adaptive Slow”). In all attack variants, the model either does not converge, or only converges when the adversary allows it to.

5.3 Poisoning attacks by sybil clones

We demonstrate the ineffectiveness of prior defenses by performing a targeted poisoning attack with sybil clones in a non-IID FL setting, where 10 clients train an MNIST 1-layer softmax classifier; each client holds a distinct digit from the original training dataset.

We perform the attack against a variety of aggregation techniques: Multi-Krum (using $f = \text{sybils}$, the best case scenario for the defense), median, trimmed-mean (using $\beta = [10\%, 20\%]$), a baseline evaluation (using the mean across clients) and FoolsGold, our proposed solution that relies on client similarity, described in detail in Section 6 (using FEDSGD and FEDAVG)¹.

¹In prior work [20], we also performed a targeted sybil-based poisoning attack against RONI [5], a defense that relies on a validation dataset, but

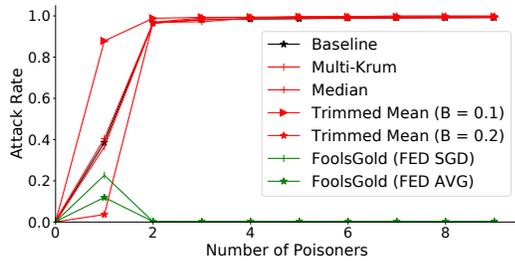


Figure 4: **Label-flipping** attack rate for varying number of sybils, for FL (Baseline), Multi-Krum, Median, Trimmed Mean and FoolsGold.

To perform the *label-flipped attack*, each sybil client holds the same dataset of 1s, all labeled as 7s. A successful label-flipping attack would produce a model that incorrectly classifies all 1s as 7s. To perform the *backdoor attack*, we use a single white pixel in the bottom-right corner as the backdoor pattern [23]. Each sybil client holds a random uniform subset of the MNIST data, where each image is marked with a white pixel in the bottom-right corner of the image, and labeled as a 7. A successful backdoor attack results in a model where all images with the backdoor inserted (white bottom-right pixel) would be predicted as a 7, regardless of the other information in the image.

For both attacks, the number of sybils executing the attack increases from 0 to 9. Figures 4 and 5 show the performance of the approaches against the label-flipping and backdoor attacks respectively, where the attack rate is defined as the proportion of targeted examples (originally labeled 1s or images with the backdoor pixel) in the test set that are misclassified as 7s (the poisoning objective).

As soon as the proportion of sybil-based poisoners for a single class increases beyond the corresponding number of honest clients that hold that class (which is 1 in this case), the attack rate increases significantly for naive averaging (labeled as “Baseline”).

The performance of Multi-Krum is especially poor in the non-IID setting. When the variance of updates among honest clients is high, and the variance of updates among sybils is lower, Multi-Krum removes honest clients from the system. Multi-Krum is unsuitable for defending FL against sybils in its intended non-IID setting. Similarly, both the median and the trimmed mean are inadequate once the number of sybils increases. In all cases, a large number of sybils will skew this summary statistic, causing FL to fail.

Summary statistics are not a viable solution to sybils.

Clearly, when sybils are present in a FL system, relying on a summary statistic (such as the mean, median, mode or any distribution-based summarizing technique) is an inadequate defense. When the number of sybils is high enough, these

found that RONI was trivially beaten by the sybil-based poisoning attack and we therefore omit these results for space.

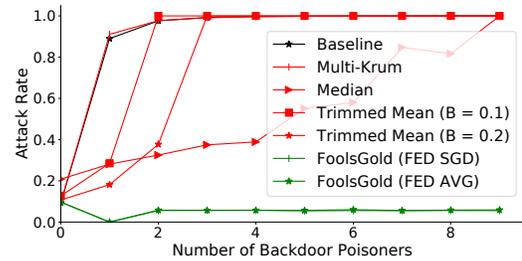


Figure 5: **Backdoor** attack rate for varying number of sybils, for FL (Baseline), Multi-Krum, Median, Trimmed Mean and FoolsGold.

statistics are manipulated by the adversary and in fact will cause honest contributions to be labeled as anomalous and removed.

FoolsGold is a defense that does not require explicit parameterization of the number of attackers. The key assumption in FoolsGold is that: when performing targeted poisoning, sybils contribute updates that appear more similar than the expected similarity found between honest clients. When an abnormally high similarity is observed, those client contributions are penalized with a lower learning rate. We further discuss the design and motivation of FoolsGold in Section 6.

In contrast to the summary statistics described above, FoolsGold penalizes attackers further as the proportion of similar updates increases, and in Figures 4 and 5 FoolsGold remains robust even with 9 sybils. Since this attack uses client-contribution similarity, FoolsGold performs the worst when defending against one poisoner. We mitigate this weakness in Section 7.7.

In the rest of the paper we focus on targeted poisoning attacks and defenses in the context of clone-based sybils.

6 FoolsGold: countering targeted sybil poisoning attacks

We now describe FoolsGold, a defense that uses client similarity to prevent sybil-based targeted poisoning attacks in FL². Unlike other defenses, FoolsGold does not require knowledge of the number of sybils, does not require modifications to the client-side protocol, and only uses state from the learning process itself. The FoolsGold algorithm and motivation are also described in our prior work [20].

6.1 FoolsGold threat model

In our setting, sybils observe global model state and send any arbitrary gradient contribution to the aggregator at any iteration. We assume that some honest clients have unpoisoned training data, requiring that every class in the model

²System implementation and experiments are available at <https://github.com/DistributedML/FoolsGold>

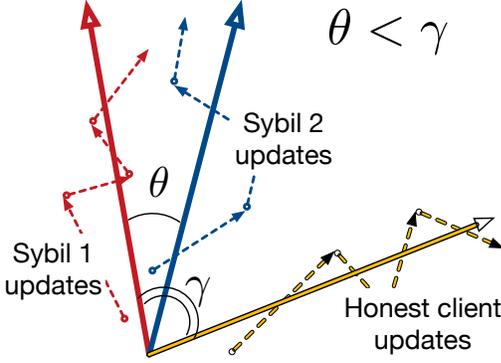


Figure 6: Dashed lines are gradient updates from three clients (2 sybils, 1 honest). Solid lines are aggregated update vectors. The angle between the aggregated update vectors of sybil clients (θ) is smaller than between those of the honest client and a sybil (γ). Cosine similarity would reflect this similarity.

is represented by at least one honest client’s dataset. Without these honest clients, no contribution-based defense is possible since the model would be unable to learn anything about these classes in the first place.

One possible attack strategy involves scaling malicious updates to overpower honest clients [3, 61]. However, since state of the art magnitude-based detection methods succeed in preventing these attacks [9, 61], we do not consider this strategy in our work.

Secure-aggregation for FL provides privacy by obfuscating client updates [11]. For any server-side defense to operate through observation of malicious updates, we must assume that these types of obfuscations are not used. We also require that FL is performed synchronously, as is assumed by most other attacks and defenses in FL [9, 19, 59, 61, 62].

Client-side differential privacy has also been proposed in FL [21]: from the server’s perspective, the aggregation rules are performed in the same way. Furthermore, since the algorithm is performed on the client device, we assume that sybils are not required to respect this protocol when performing attacks.

6.2 FoolsGold design

In the FL protocol (Algorithm 1), gradient updates are collected and aggregated in synchronous update rounds³.

When each client’s training data is non-IID and has a unique distribution, we assume that *honest clients can be distinguished from act-alike sybils by the diversity of their gradient updates*: sybils will contribute updates that appear more similar to each other than those among honest clients. Although this assumption is most clear in the federated non-IID setting, FoolsGold does not rely on the data being non-

³From the aggregator perspective, this is true regardless of FEDAVG or FEDSGD. We show that FoolsGold can be applied in both settings.

Data: Initial Model w_0 and SGD updates $\Delta_{i,t}$ from each client i at iteration t . Confidence parameter κ

```

1 for Iteration  $T$  do
2   // Per client learning rate for iteration  $T$ 
3   Initialize  $\alpha$ 
4   for All clients  $i$  do
5     // Updates history
6     Let  $H_i$  be the aggregate historical vector  $\sum_{t=1}^T \Delta_{i,t}$ 
7     // Feature importance
8     Let  $S_T$  be the weight of important features at iteration  $T$ 
9     for All other clients  $j$  do
10      Let  $cs_{ij}$  be the  $S_T$ -weighted cosine similarity
11      between  $H_i$  and  $H_j$ 
12    end
13    Let  $v_i = \max_j(cs_{ij})$ 
14  end
15  for All clients  $i$  do
16    for All clients  $j$  do
17      // Pardoning
18      if  $v_j > v_i$  then
19         $cs_{ij} *= v_i/v_j$ 
20      end
21    end
22    // Per-row maximums
23     $\alpha_i = 1 - \max_j(cs_{ij})$ 
24  end
25  // Normalize learning rates to 0-1 range
26   $\alpha = \alpha / \max_i(\alpha_i)$ 
27  // Element-wise logit function
28   $\alpha = \kappa(\ln[(\alpha)/(1 - \alpha)] + 0.5)$ 
29  // Federated SGD iteration
30   $w_T = w_{T-1} + \sum_i \alpha_i \Delta_i$ 
31 end

```

Algorithm 1: FoolsGold learning algorithm.

IID; we also explore FoolsGold’s performance in varying IID settings in Section 7.2.

FoolsGold adapts the learning rate α_i per client⁴ based on (1) the update similarity among indicative features in any given iteration, and (2) historical information from past iterations.

Cosine similarity. We use cosine similarity to measure the angular distance between updates. This is preferred to Euclidean distance since sybils can manipulate the magnitude of a gradient to achieve dissimilarity, but the direction of a gradient is indicative of the update’s objective.

Feature importance. From the perspective of a poisoning attack, there are three types of features in the model: (1) features that are relevant to the correctness of the model, but must be modified for a successful attack, (2) features that are relevant to the correctness of the model, but irrelevant for the attack, and (3) features that are irrelevant to both the attack

⁴Note that we use α for the FoolsGold assigned learning rate, and η for the traditional, local learning rate. These are independent of each other.

and the model effectiveness.

Similar to other decentralized poisoning defenses [50], we look for similarity only in the indicative features (type 1 and 2) in the model. This prevents adversaries from manipulating irrelevant features while performing an attack, which is evaluated in Section 7.3.

The indicative features are found by measuring the magnitude of model parameters in the output layer of the global model, since this maps directly to its influence on the prediction probability [51]. The updates on these features can be removed based on a threshold (hard) or re-weighted based on their influence on the model (soft), and are normalized across all classes.

For deep neural networks, we do not consider the magnitude of values in the non-output layers of the model, which do not map directly to output probabilities and are more difficult to reason about. Recent work on feature influence in deep neural networks [1, 16, 32] may better capture the intent of sybil-based poisoning attacks in deep learning and we leave this analysis as future work.

Update history. FoolsGold maintains a history of updates from each client by aggregating the updates over multiple iterations (line 4). To better estimate client similarity, FoolsGold computes the pairwise similarity between aggregated historical updates instead of the updates from just the current iteration.

Figure 6 shows that even for two sybils with a common target objective, updates at a given iteration may diverge due to the variance of SGD. However, the cosine similarity between the sybils’ aggregated historical updates tends to converge towards the malicious objective, providing a more accurate estimate of the client intent.

We interpret the cosine similarity on the indicative features, a value between -1 and 1, as a representation of how strongly two clients are acting as sybils. We define v_i as the maximum pairwise similarity for a client i , ensuring that as long as one such interaction exists, we can devalue the contribution while staying robust to an increasing number of sybils.

Pardoning. Since we have weak guarantees on the cosine similarities between an honest client and sybils, honest clients may be incorrectly penalized under this scheme. We introduce a pardoning mechanism that avoids penalizing such honest clients by re-weighting the cosine similarity by the ratio of v_i and v_j (line 14), reducing false positives. The new client learning rate α_i is then found by inverting the maximum similarity scores along the 0-1 domain. Since we assume at least one client in the system is honest, we rescale the vector such that the maximum adaption of the learning rate is 1 (line 19). This ensures that at least one client will have an unmodified update and encourages that a system with only honest nodes will not penalize their contributions.

Logit. However, even for very similar updates, the cosine similarity may be less than one. An attacker may exploit this by increasing the number of sybils to remain influential. We

Dataset	Examples	Classes	Features	Model
MNIST	60,000	10	784	1-layer
VGGFace2	7,380	10	150,528	ImageNet
KDDCup	494,020	23	41	1-layer
Amazon	1,500	50	10,000	1-layer

Table 4: Datasets used in this evaluation.

therefore want to encourage a higher divergence for values that are near the tails of this function, and avoid penalizing honest clients with a low, non-zero similarity value. Thus, we use the logit function (the inverse sigmoid function) centered at 0.5 (line 20), to encourage these properties. We also expose a confidence parameter κ that scales the logit function and show in Appendix A that κ can be set as a function of the expected data distribution among clients.

When taking the result of the logit function, any value exceeding the 0-1 range is clipped and rounded to its respective boundary value. Finally, the gradient update is calculated by applying the final re-scaled learning rate to the global model.

7 FoolsGold evaluation

We evaluate FoolsGold on a federated learning prototype implemented in 600 lines of Python. The prototype includes 150 lines for FoolsGold, implementing Algorithm 1. We use scikit-learn [46] to compute cosine similarity of vectors. For each experiment below, we partition the original training data into disjoint training sets, locally compute SGD updates on each dataset, and aggregate the updates using the described FoolsGold method to train a globally shared classifier.

We evaluate our prototype on four classification datasets (described in Table 4): MNIST [31], a digit classification problem, VGGFace2 [14], a facial recognition problem, KDDCup [17], which contains classified network intrusion patterns, and Amazon [17], which contains text from product reviews.

Each dataset was selected for one of its particularities. MNIST was chosen as the baseline dataset for evaluation since it was used extensively in the original federated learning evaluation [39]. The VGGFace2 dataset was chosen as a more complex learning task that requires deep neural networks to solve. For simplicity in evaluating poisoning attacks, we limit this dataset to the top 10 most frequent classes only. The KDDCup dataset has a relatively low number of features, and contains a massive class imbalance: some classes have as few as 5 examples, while some have over 280,000. Lastly, the Amazon dataset is unique in that it has few examples and contains text data: each review is one-hot-encoded, resulting in a large feature vector of size 10,000.

For all the experiments in this section, we perform targeted poisoning attacks that attempt to encourage a *source label/pattern* to be classified as a *target label* while training

through federated learning⁵. In our baseline experiments, each class is solely owned by a single client, which is consistent with the federated learning baseline. In all experiments the number of honest clients matches the number of classes used in the dataset: 10 for MNIST and VGGFace2, 23 for KDD-Cup, and 50 for Amazon. For more-IID settings, we modify the distribution of client data and consider settings where classes overlap between clients in Section 7.2.

For MNIST, KDDCup, and Amazon, we train a 1-layer softmax classifier. For VGGFace2, we use two popular pre-trained Imagenet architectures from the torchvision package [38]: SqueezeNet1.1, a compressed model of 727,000 parameters designed for edge devices; and VGGNet11, a larger model of 128,000,000 parameters. When comparing client similarity for FoolsGold, we only use the features in the final output layer’s gradients (fully connected layer in VGGNet and 10 1x1 convolutional kernels in SqueezeNet).

In MNIST, the data is already divided into 60,000 training examples and 10,000 test examples [31]. For VGGFace2, KDDCup and Amazon, we randomly partition 70% of the total data as training data and 30% as test data. The test data is used to evaluate two metrics that represent the performance of our algorithm: the *attack rate*, which is the proportion of attack targets (source labels/patterns) that are incorrectly classified as the target label, and the *test accuracy*, which is the proportion of examples in the test set that are correctly classified.

The MNIST and KDDCup datasets were executed with 3,000 iterations and a batch size of 50 unless otherwise stated. For Amazon, due to the high number of features and low number of samples per class, we train for 100 iterations and a batch size of 10. For VGGFace2, we train for 500 iterations with batch size of 8, momentum 0.9, weight decay 0.0001, and learning rate 0.001. These values were found using cross validation in the training set. During training, images were resized to 256x256 and randomly cropped and flipped to 224x224. During testing, images were resized to 256x256 and a 224x224 center was cropped.

We showed FoolsGold’s effectiveness both when FEDSGD and FEDAVG are used by the clients. Since the difference between FEDSGD and FEDAVG was negligible in Figures 4 and 5, we continued to use FEDSGD for all future experiments.

We report the mean across 5 experiments in all cases. For each experiment, FoolsGold is parameterized with a confidence parameter $\kappa = 1$, and does not use the historical gradient or the significant features filter (we evaluate these design elements independently in Section 7.3 and 7.5, respectively).

⁵For the MNIST, VGGFace2, and Amazon datasets, we evaluated all source/target class pairs and found that the performance difference between these attacks was marginal.

Attack	Description	Dataset
A-1	1 sybil attacks.	All
A-5	5 sybils attack.	All
A-5x5	5 sets of 5 sybils, concurrent attacks.	MNIST, Amazon, VGGFaces2
A-OnOne	5 sybils executing 5 attacks on the same target class.	KDDCup99
A-99	99% sybils, same attack.	MNIST

Table 5: Canonical attacks used in our evaluation.

7.1 Canonical attack scenarios

Our evaluation uses a set of 5 canonical attack scenarios across the four datasets (Table 5). Attack **A-1** is a traditional poisoning attack: a single client joins the federated learning system with poisoned data. Attack **A-5** is the same attack performed with 5 sybil clients in the system. Each client sends updates for a subset of its data through SGD, meaning that their updates are not identical. Attack **A-5x5** evaluates FoolsGold’s ability to thwart multiple attacks at once: 5 sets of client sybils attack the system concurrently, and we assume that the classes in these attacks do not overlap⁶.

Since KDDCup99 is a unique dataset with severe class imbalance, instead of using an A-5x5 attack we choose to perform a different attack, **A-OnOne**. In KDDCup99, data from various network traffic patterns are provided. Class ‘Normal’ identifies patterns without any network attack, and is proportionally large (~ 20% of the data). When attacking KDDCup99, we assume that adversaries mislabel malicious attack patterns, which are proportionally small, (on average ~ 2% of the data) and poison the malicious class towards the ‘Normal’ class. A-OnOne is a unique attack for KDDCup in which 5 different malicious patterns are each labeled as ‘Normal’, and each attack is performed concurrently.

Finally, we use **A-99** to illustrate the robustness of FoolsGold to a massively powerful adversary that generates 990 sybils to overpower a network of 10 honest clients, all of them performing the same attack against MNIST.

Since we use these canonical attacks throughout this work, we first evaluate FoolsGold against each attack on their respective datasets and models. Figure 7 plots the attack rate and test accuracy for each attack in Table 5. We also show results for the system without attacks: the original federated learning algorithm (FL-NA) and the system with the FoolsGold algorithm (FG-NA).

Figure 7 shows that for most attacks, FoolsGold effectively prevents the attack while maintaining high test accuracy. As FoolsGold faces larger groups of sybils, it has more information to more reliably detect similarity between sybils. FoolsGold performs worst on the A-1 attacks in which only one malicious client attacks the system. This is expected; without

⁶We do not perform a 1-2 attack in parallel with a 2-3 attack, since evaluating the 1-2 attack would be biased by the performance of the 2-3 attack.

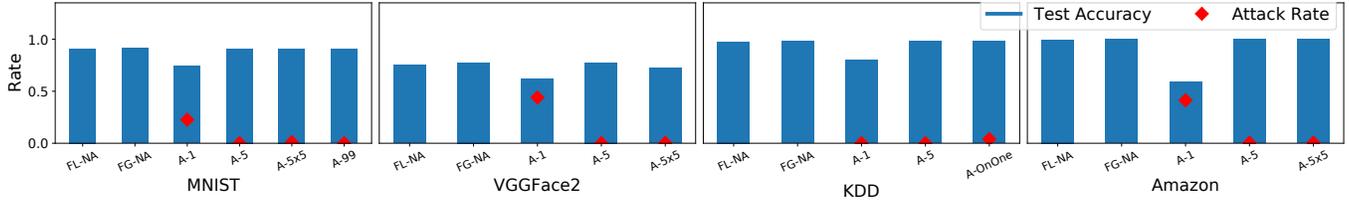


Figure 7: Test accuracy (blue bars) and attack rate (red ticks) for canonical attacks against the relevant canonical datasets.

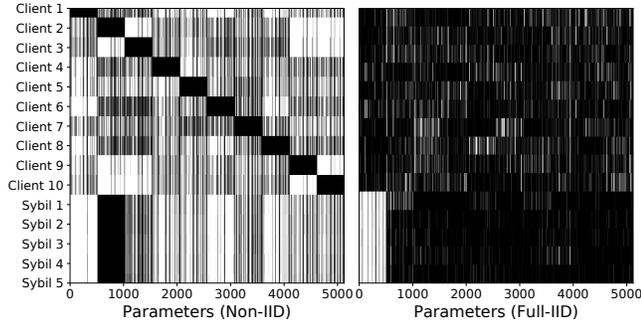


Figure 8: A visualization of the historical gradient of each clients on the Squeezenet model, in the non-IID (left) and IID (right) case. The top 10 rows show the gradient vector of honest clients; the bottom 5 rows for sybils performing a targeted 0-1 attack.

multiple colluding sybils, malicious and honest clients are indistinguishable to FoolsGold.

Another point of interest is the prevalence of false positives. In A-1 KDDCup, our system incorrectly penalized an honest client for colluding with the attacker, lowering the prediction rate of the honest client as the defense was applied. We observe that the two primary reasons for low test accuracy are either a high attack rate (false negatives) or a high target class error rate (false positives). We also discuss false positives from data similarity in Section 7.2.

7.2 FoolsGold on varying IID settings

By design, FoolsGold relies on the assumption that training data is sufficiently dissimilar between clients. However, a more realistic scenario may involve settings where local client data distributions overlap more significantly.

To understand how FoolsGold handles these situations, we execute a VGGFace2 A-5 experiment under varying client distributions. Figure 8 shows each client’s FoolsGold vector after 3000 training iterations, where each row shows the flattened historical gradient of a client on their final softmax layer. The top 10 rows correspond to honest clients and the bottom 5 rows correspond to sybils. The left half of Figure 8 shows

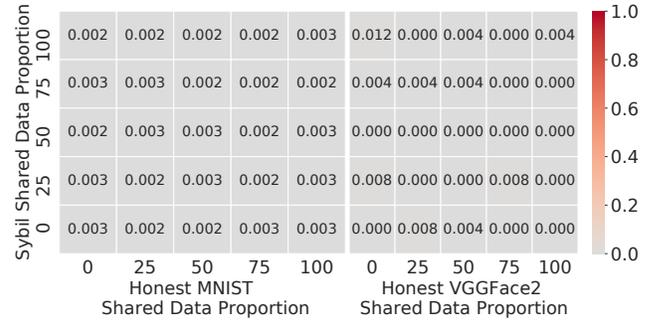


Figure 9: The attack rates on the MNIST (with softmax model) and VGGFace2 (with SqueezeNet model) datasets for varying sybil and honest client data distributions. 0 means that the data is completely disjoint by class; 100 means that the data is uniformly distributed.

the historical vector $\sum_{t=0}^T \Delta_{i,T}$ in a non-IID setting. Since each honest client has data corresponding to a unique class, their gradients are highly dissimilar, and since the sybils have the same poisoning dataset for a 0-1 attack, their gradients are highly similar. Therefore, FoolsGold can easily detect the sybil gradients.

The right half of Figure 8 shows the historical vector $\sum_{t=0}^T \Delta_{i,T}$ in a full-IID setting. Despite the honest clients holding uniform samples of the training data, the stochasticity of SGD introduces variance between them. The sybil gradients are still uniquely distinct as a result of their poisoned data, as seen in the bottom left corners of Figure 8. Even with only 10% poisoned data, executing a targeted poisoning attack produces fundamentally similar gradient updates in both full-IID and non-IID settings, enabling FoolsGold to succeed.

To test FoolsGold under diverse data distribution assumptions, we conduct an A-5 0-1 attack (with 5 sybils) on MNIST and VGGFace2 with 10 honest clients while varying the proportion of shared labels between both the sybils and honest clients. We varied these proportions over a grid ($D_{sybil}, D_{honest} \in \{0, 0.25, 0.5, 0.75, 1\}$), where D refers to the ratio of disjoint data to shared data. $D = 0$ refers to a non-IID setting where each client’s local dataset is composed of a single class, and $x = 1$ refers to a setting where each client’s local

dataset is uniformly sampled from all classes. For all other cases, clients hold a proportion of D_{honest} uniform data and $(1 - D_{honest})$ non-IID data from a single class. For sybils, we first create an honest client using the above mechanism, and flip all 0 labels to a 1 to perform a targeted attack. Therefore, when $D_{sybil} = 0$, sybils will hold a full dataset of 0-1 poisoned data, and when $D_{sybil} = 1$, sybils will hold a dataset of 10% poisoned 0-1 data, and 90% uniform data from classes 1-9.

FoolsGold defends against poisoning attacks for all (D_{sybil}, D_{honest}) combinations: the maximum attack rate was less than 1% for both the MNIST and VGGFace2 datasets, using both SqueezeNet and VGGNet. We show these results in Figure 9. In summary, an attacker cannot subvert FoolsGold by manipulating their malicious data distribution⁷. Instead, they must directly manipulate their gradient outputs, which we explore next.

If an attacker is aware of the FoolsGold algorithm, they may attempt to send updates in ways that encourage additional dissimilarity. This is an active trade-off: as attacker updates become less similar to each other (lower chance of detection), they become less focused towards the poisoning objective (lower attack utility).

Next, we consider and evaluate two synchronized sybil strategies in which attackers may subvert FoolsGold: (1) perturbing contributed updates to maximize dissimilarity, and (2) infrequently and adaptively sending poisoned updates.

7.3 Attacks using intelligent perturbations

A set of intelligent sybils could synchronize and send pairs of updates with careful perturbations that are designed to sum to zero. For example, if an attacker draws a perturbation vector ζ , two malicious updates a_1 and a_2 could be contributed as v_1 and v_2 , such that $v_1 = a_1 + \zeta$ and $v_2 = a_2 - \zeta$.

Since the perturbation vector ζ has nothing to do with the poisoning objective, its inclusion will add dissimilarity to the malicious updates and decrease FoolsGold’s effectiveness in detecting them. Also note that the sum of these two updates is still the same: $v_1 + v_2 = a_1 + a_2$. This strategy can also be scaled beyond 2 sybils by taking orthogonal perturbation vectors and their negation: for any subset of these vectors, the cosine similarity is 0 or -1, while the sum remains 0.

As explained in Section 6, this attack is most effective if ζ is only applied to features of type (3): those which are not important for the model or the attack. The attack is mitigated by filtering for indicative features in the model. Instead of looking at the cosine similarity between updates across all features in the model, we look at a weighted cosine similarity based on feature importance.

To evaluate the importance of this mechanism to the poisoning attack, we execute the intelligent perturbation attack

⁷In prior work [20], we also evaluated FoolsGold against other method for increased SGD diversity: mixing honest data with malicious data and using settings with a decreased SGD batch size.

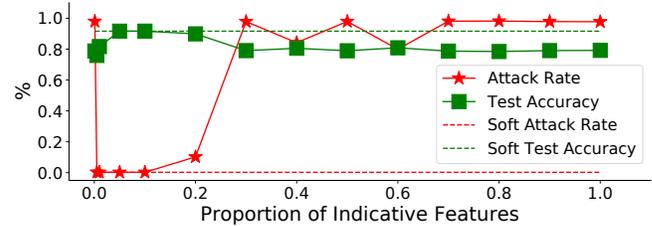


Figure 10: The performance of the optimal perturbation attack on MNIST with varying indicative feature ratio.

described above on MNIST, which contains several irrelevant features (the black background) in each example: a pair of sybils send v_1 and v_2 with intelligent perturbation ζ . We then vary the proportion of model parameters that are defined as indicative from 0.001 (only top 8 features on MNIST) to 1 (all features).

Figure 10 shows the attack rate and the test accuracy for varying proportions of indicative features against an A-5 attack. We first observe that when using all of the features for similarity (far right), the poisoning attack is successful.

Once the proportion of indicative features decreases below 0.1 (10%), the dissimilarity caused by the intelligent perturbation is removed from the cosine similarity and the poisoning vector dominates the similarity, causing the intelligent perturbations strategy to fail with an attack rate of near 0. We also observe that if the proportion of indicative features is too low (0.01), the test accuracy also begins to suffer. When considering such a low number of features, honest clients appear to collude as well, causing false positives.

We also evaluated the soft feature weighing mechanism, which weighs each contribution proportionally based on the model parameter itself. The results of the soft weighting method on the same intelligent MNIST poisoning attack are also shown in Figure 10. For both the attack rate and test accuracy, the soft filtering mechanism is comparable to the optimal performance of the hard filtering mechanism.

7.4 Attacks with adaptive updates

We devised another synchronized attack against FoolsGold that manipulates its memory component. If an adversary knows that FoolsGold uses similarity on the update history, and is able to locally compute its own pairwise cosine similarity among sybils, they can collude and compute this information themselves, deciding only to send poisoned updates when their historical similarity is low. We define a parameter M for the attack strategy that represents the threshold on inter-sybil similarity for sybils to send a poisoned update. When M is lower, sybils are less likely to be detected by FoolsGold and will send their updates less often; however, this will also lower the influence the sybils have on the global model.

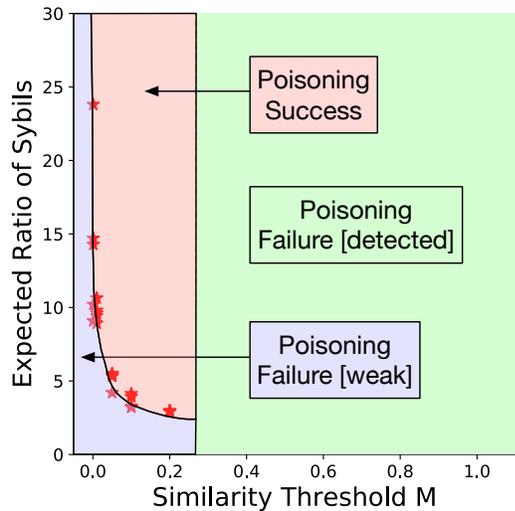


Figure 11: Relationship between similarity threshold and expected ratio of sybils per honest opposing client for the synchronized adaptive attack strategy against FoolsGold with 2 sybils on MNIST.

An adversary could generate an excess number of sybils for a successful attack, but given that the adversary is uncertain about the influence needed to overpower the honest clients, this is a difficult trade-off to predict for an optimal attack.

To demonstrate this, the intelligent perturbation attack above is executed by 2 sybils on MNIST, with FoolsGold using the soft weighing of features in its cosine similarity (the optimal defense for MNIST against the intelligent perturbation attack). Figure 11 shows the relationship between M and the resulting expected ratio of sybils needed to match the influence for each honest opposing client.

For instance, if we observed that the sybils only sent poisoning gradients 25% of the time, they would need 4 sybils to induce a comparable influence on the model. Given a prescribed similarity threshold M , the values shown are the expected number of sybils required for the optimal attack. The attack is optimal because using less sybils does not provide enough influence to poison the model, while using more sybils is inefficient.

This is shown in Figure 11 with three shaded regions: in the green region to the right ($M > 0.27$), the threshold is too high and any poisoning attack is detected and removed. In the blue region on the bottom left, the attack is not detected, but there is an insufficient number of sybils to overpower the honest opposing clients. Lastly, in the top left red region, the attack succeeds, potentially with more sybils than required.

With a sufficiently large number of sybils and appropriately low threshold, attackers can subvert our current defense for our observed datasets. Although this strategy can break FoolsGold, finding the appropriate threshold is challenging as it is dependent on many other factors: the number of honest

clients in the system, the proportion of indicative features considered by FoolsGold, and the distribution of data. The exact number of sybils required to successfully poison the model is unknown to attackers without knowledge of the number of honest clients and their honest training data.

7.5 Effects of design elements

Each of the three main design elements (history, pardoning and logit) described in Section 6 addresses specific challenges. In the following experiments we disabled one of the three components and recorded the test error, attack rate, and target class test error of the resulting model.

History. Attacks with intelligent perturbations and adaptive updates increase the variance of updates in each iteration. The increased variance in the updates sent by sybils cause the cosine similarities at each iteration to be an inaccurate approximation of a client’s sybil likelihood. Our design uses history to address this issue, and we evaluate it by comparing the performance of FoolsGold with and without history using an A-5 MNIST attack with 80% poisoned data and batch size of 1 (factors which will induce a high variance).

Pardoning. We claim that honest client updates may be similar to the updates of sybils, especially if the honest client owns the data for the targeted class. To evaluate the necessity and efficacy of our pardoning system, we compare the performance of FoolsGold on KDDCup with the A-AllOnOne attack with and without pardoning.

Logit. An important motivation for using the logit function is that adversaries could otherwise arbitrarily increase the number of sybils to mitigate any non-zero weighting of their updates. We evaluate the performance of FoolsGold with and without the logit function for the A-99 MNIST attack.

Figure 12 shows the overall test error, sybil attack rate, and target class test error for the six different evaluations. The attack rate for the A-AllOnOne KDDCup attack is the average attack rate for the 5 sets of sybils.

Overall, the results align with our claims. For the A-5 MNIST case, we find that history successfully mitigates attacks that otherwise would pass through in the no-history system. Comparing the results of the A-AllOnOne KDDCup attack, we find that, without pardoning, the test error of both the target class and the overall test error increase while the attack rate was negligible for both cases, indicating a high rate of false positives for the target class. Finally, for the A-99 MNIST attack, without the logit function, the adversary was able to mount a successful attack by overwhelming FoolsGold with sybils, showing that the logit function is necessary to prevent this attack.

7.6 FoolsGold performance overhead

We evaluate the runtime overhead incurred by augmenting a federated learning system with FoolsGold. We run the system

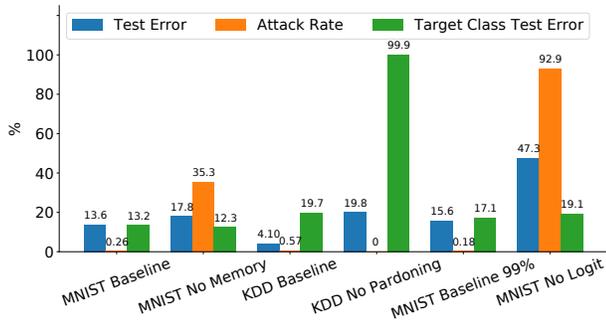


Figure 12: Metrics for FoolsGold with various components independently removed.

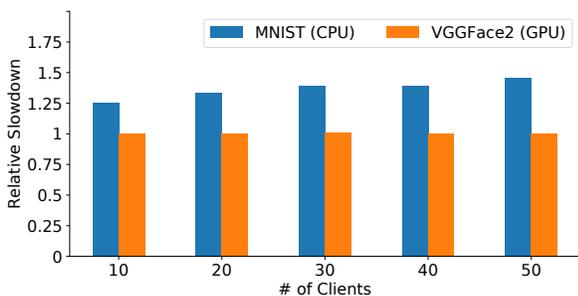


Figure 13: Running time overhead of FoolsGold as compared to Federated Learning (baseline of 1), for MNIST (on a CPU) and VGGFace2 (on a GPU).

with and without FoolsGold with 10 – 50 clients by training an MNIST classifier on a commodity CPU and a VGGFace2 deep learning model on a Titan V GPU.

Figure 13 plots the relative slowdown added by FoolsGold for CPU and GPU based workloads. On a CPU, the most expensive part of the FoolsGold algorithm is computing the pairwise cosine similarity. Our Python prototype is not optimized and there are known optimizations to improve the speed of computing angular distance at scale [2]. When training a deep learning model on a GPU, the cost of training is high enough that the relative slowdown from FoolsGold is negligible. We profiled micro-benchmarks for the total time taken to execute the FoolsGold algorithm and found that it took less than 1.5 seconds, even with 50 clients.

7.7 Combating a single client adversary

We consider the single-shot model replacement attack [3] in which a single adversarial client sends the direct vector to the poisoning objective. This attack does not require sybils and can therefore bypass FoolsGold.

We performed an experiment that augmented FoolsGold with a properly parameterized Multi-Krum solution, with $f = 1$. Figure 14 shows the training accuracy and the attack rate

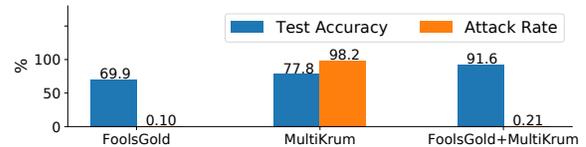


Figure 14: The performance of Multi-Krum with FoolsGold when combating a combination of an A-5 attack and the model replacement attack [3] on MNIST.

for FoolsGold, Multi-Krum, and the two systems combined when facing concurrent A-5 and model replacement attacks.

We see that Multi-Krum and FoolsGold do not interfere with each other. The Multi-Krum algorithm prevents the model replacement attack, and FoolsGold prevents the sybil attack. Independently, these two systems fail to defend both attacks concurrently, either by failing to detect the model replacement attack (for FoolsGold) or by allowing the sybils to overpower the system (for Multi-Krum).

FoolsGold is specifically designed for handling targeted poisoning attacks from a group of clone-based sybils: we believe the current state of the art is better suited to mitigate attacks from single actors.

8 Conclusion

The decentralization of ML is driven by growing privacy and scalability challenges. Federated learning is a state of the art proposal adopted in production [40], and is increasingly being used in mobile and edge networks [33]. However, using federated learning in such settings has opened the door for adversaries to attack the system with sybils [44]. We showed that federated learning is vulnerable to sybil-based attacks and that existing defenses are ineffective. To defend against one of these strategies (sybil-based clones that remain in the system), we proposed *FoolsGold*, a defense that uses client *contribution similarity*. Our results indicate that FoolsGold mitigates targeted poisoning attacks and is effective even when sybils overwhelm the honest clients.

Despite FoolsGold’s performance against targeted poisoning attacks, a number of sybil-based strategies are not well defended by FoolsGold, such as coordinated attacks, some of which we showed (adaptive attacks, intelligent perturbations) and some of which have been proposed in recent work (distributed backdoors [58]).

In addition, we suggest that there is a much higher potential for sybils to be used to execute attacks on distributed multi-party ML systems such as federated learning. We hope that our work inspires further research on the implications of sybils on these systems and leads to other defenses that are robust to sybils.

Acknowledgments

This work was supported by the Huawei Innovation Research Program (HIRP), Project No: HO2018085305. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 2014-04870. Chris contributed to the project while being sponsored by the NSERC USRA program.

References

- [1] M. Ancona, E. Ceolini, A. C. Oztireli, and M. Gross. A Unified View of Gradient-based Attribution Methods for Deep Neural Networks. In *Workshop on Interpreting, Explaining and Visualizing Deep Learning*, NIPS, 2017.
- [2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems 28*, NIPS, 2015.
- [3] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How To Backdoor Federated Learning. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, AISTATS, 2020.
- [4] Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, and Jaehoon Amir Safavi. Mitigating Poisoning Attacks on Machine Learning Models: A Data Provenance Based Approach. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec, 2017.
- [5] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The Security of Machine Learning. *Machine Learning*, 81(2), 2010.
- [6] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing Federated Learning through an Adversarial Lens. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 634–643. PMLR, 2019.
- [7] Battista Biggio, Iginio Corona, Giorgio Fumera, Giorgio Giacinto, and Fabio Roli. Bagging Classifiers for Fighting Poisoning Attacks in Adversarial Classification Tasks. In *Multiple Classifier Systems*, 2011.
- [8] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning Attacks Against Support Vector Machines. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML, 2012.
- [9] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *Advances in Neural Information Processing Systems 30*, NIPS, 2017.
- [10] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé M Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards Federated Learning at Scale: System Design. In *Conference on Systems and Machine Learning*, SysML, 2019.
- [11] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.
- [12] Léon Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In *19th International Conference on Computational Statistics*, COMPSTAT, 2010.
- [13] Sebastian Caldas, Jakub Konečný, H Brendan McMahan, and Ameet Talwalkar. Expanding the Reach of Federated Learning by Reducing Client Resource Requirements. *arXiv preprint arXiv:1812.07210*, 2019.
- [14] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman. VGGFace2: A Dataset for Recognising Faces across Pose and Age. In *International Conference on Automatic Face and Gesture Recognition*, FG, 2018.
- [15] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *arXiv preprint arXiv:1712.05526*, 2017.
- [16] A. Datta, S. Sen, and Y. Zick. Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, S&P, 2016.
- [17] Dua Dheeru and Efi Karra Taniskidou. UCI Machine Learning Repository, 2017.
- [18] John (JD) Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, IPTPS, 2002.
- [19] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local Model Poisoning Attacks to Byzantine-Robust Federated Learning. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [20] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. Mitigating Sybils in Federated Learning Poisoning. *arXiv preprint arXiv:1808.04866*, 2018.
- [21] Robin C. Geyer, Tassilo Klein, and Moin Nabi. Differentially Private Federated Learning: A Client Level Perspective. *NIPS Workshop: Machine Learning on the Phone and other Consumer Devices*, 2017.
- [22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, 2017.
- [23] T. Gu, B. Dolan-Gavitt, and S. Garg. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *arXiv preprint arXiv:1708.06733*, 2017.
- [24] J. Hamm, A. C. Champion, G. Chen, M. Belkin, and D. Xuan. Crowd-ML: A Privacy-Preserving Learning Framework for a Crowd of Smart Devices. In *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems*, ICDCS, 2015.
- [25] Bo Han, Ivor W. Tsang, and Ling Chen. On the Convergence of a Family of Robust Losses for Stochastic Gradient Descent. In *European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 9851*, ECML PKDD, 2016.

- [26] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2017.
- [27] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed Machine Learning Approaching LAN Speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation, NSDI 17*, 2017.
- [28] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. Adversarial Machine Learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, AISec*, 2011.
- [29] Linshan Jiang, Rui Tan, Xin Lou, and Guosheng Lin. On Lightweight Privacy-preserving Collaborative Learning for Internet-of-things Objects. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI*, 2019.
- [30] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and Open Problems in Federated Learning. *arXiv preprint arXiv:1912.04977*, 2019.
- [31] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 1998.
- [32] K. Leino, S. Sen, A. Datta, M. Fredrikson, and L. Li. Influence-Directed Explanations for Deep Convolutional Networks. In *2018 IEEE International Test Conference, ITC*, 2018.
- [33] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. Federated Learning in Mobile Edge Networks: A Comprehensive Survey. *arXiv preprint arXiv:1909.11875*, 2019.
- [34] Jierui Lin, Min Du, and Jian Liu. Free-riders in Federated Learning: Attacks and Defenses. *arXiv preprint arXiv:1911.12560*, 2019.
- [35] Lingjuan Lyu, Han Yu, and Qiang Yang. Threats to Federated Learning: A Survey. *arXiv preprint arXiv:2003.02133*, 2020.
- [36] Saeed Mahloujifar, Mohammad Mahmood, and Ameer Mohammed. Multi-party Poisoning through Generalized p -Tampering. *arXiv preprint arXiv:1809.03474*, 2018.
- [37] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. Early Stopping without a Validation Set. *arXiv preprint arXiv:1703.09580*, 2017.
- [38] Sébastien Marcel and Yann Rodriguez. Torchvision the Machine-vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia, MM*, 2010.
- [39] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 2017.
- [40] H. Brendan McMahan and Daniel Ramage. Federated Learning: Collaborative Machine Learning without Centralized Training Data. <https://research.googleblog.com/2017/04/federated-learning-collaborative.html>, 2017.
- [41] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Exploiting Unintended Feature Leakage in Collaborative Learning. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, S&P, 2019.
- [42] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive Privacy Analysis of Deep Learning: Stand-alone and Federated Learning under Passive and Active White-box Inference Attacks. *arXiv preprint arXiv:1812.00910*, 2018.
- [43] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM J. on Optimization*, 19:1574–1609, 2009.
- [44] J. Newsome, E. Shi, D. Song, and A. Perrig. The Sybil attack in Sensor Networks: Analysis & Defenses. In *Third International Symposium on Information Processing in Sensor Networks, IPSN*, 2004.
- [45] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [47] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.
- [48] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-leak: Data Set Inference and Reconstruction Attacks in Online Learning. *arXiv preprint arXiv:1904.01067*, 2019.
- [49] Sumudu Samarakoon, Mehdi Bennis, Walid Saad, and Merouane Debbah. Distributed Federated Learning for Ultra-Reliable Low-Latency Vehicular Communications. *arXiv preprint arXiv:1807.08127*, 2019.
- [50] Shiqi Shen, Shruti Tople, and Prateek Saxena. Auror: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC*, 2016.
- [51] Reza Shokri and Vitaly Shmatikov. Privacy-Preserving Deep Learning. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2015.
- [52] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H. Brendan McMahan. Can You Really Backdoor Federated Learning? In *2nd International Workshop on Federated Learning for Data Privacy and Confidentiality, FL - NeurIPS*, 2019.
- [53] Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient Online Content Voting. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2009.

- [54] Stacey Truex, Ling Liu, Mehmet Emre Gursoy, Lei Yu, and Wenqi Wei. Towards Demystifying Membership Inference Attacks. *arXiv preprint arXiv:1807.09173*, 2018.
- [55] Bimal Viswanath, Muhammad Ahmad Bashir, Muhammad Bilal Zafar, Simon Bouget, Saikat Guha, Krishna P. Gummadi, Aniket Kate, and Alan Mislove. Strength in Numbers: Robust Tamper Detection in Crowd Computations. In *Proceedings of the 3rd ACM Conference on Online Social Networks, COSN*, 2015.
- [56] Lixu Wang, Shichao Xu, Xiao Wang, and Qi Zhu. Eavesdrop the Composition Proportion of Training Labels in Federated Learning. *arXiv preprint arXiv:1910.06044*, 2019.
- [57] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. Adaptive Federated Learning in Resource Constrained Edge Computing Systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.
- [58] Chulin Xie, Keli Huang, Pin-Yu Chen, and Bo Li. DBA: Distributed Backdoor Attacks against Federated Learning. In *International Conference on Learning Representations, ICLR*, 2020.
- [59] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Fall of Empires: Breaking Byzantine-tolerant SGD by Inner Product Manipulation. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence, UAI*, 2019.
- [60] Cong Xie, Sanmi Koyejo, and Indranil Gupta. SLSGD: Secure and Efficient Distributed On-device Machine Learning. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, ECMLPKDD*, 2019.
- [61] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Zeno: Distributed Stochastic Gradient Descent with Suspicion-based Fault-tolerance. In *Proceedings of the 36th International Conference on Machine Learning, ICML*, 2019.
- [62] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.
- [63] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. SybilLimit: A Near-optimal Social Network Defense Against Sybil Attacks. TON, 2010.
- [64] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *SIGCOMM*, 2006.

A Convergence analysis

Theorem: Given the process in Algorithm 1, the convergence rate of the participants (malicious and honest) is $O(\frac{1}{T^2})$ over T iterations.

Proof of theorem: We know from the convergence analysis of SGD [43] that for a constant learning rate, we achieve a $O(\frac{1}{T^2})$ convergence rate.

Let M be the set of malicious clients in the system and G be the set of honest clients in the system. Assume that the

adapted learning rates provided at each iteration α_i are provided by a function $h(i, t)$, where i is the client index and t is the current training iteration. As long as $h(i, t)$ does not modify the local learning rate of the honest clients and removes the contributions of sybils, the convergence analysis of SGD applies as if the training was performed with the honest clients' data.

$$\forall i \in M, h(i, t) \rightarrow 0 \quad (\text{cond1})$$

$$\forall i \in G, h(i, t) \rightarrow 1 \quad (\text{cond2})$$

We will show that, under certain assumptions, FoolsGold satisfies both conditions of $h(i, t)$. We prove each condition separately.

Condition 1: Let v_i be the ideal gradient for any given client i from the initial shared global model w_0 , that is: $w_0 + v_i = w_i^*$ where w_i^* is the optimal model relative to any client i 's local training data. Since we have defined all sybils to have the same poisoning goal, all sybils will have the same ideal gradient, which we define to be v_m .

As the number of iterations in FoolsGold increases, the historical gradient $H_{i,t}$ for each sybil approaches v_m , with error from the honest client contributions ϵ :

$$\forall i \in M : \lim_{t \rightarrow \infty} H_{i,t} = v_m + \epsilon$$

Since the historical update tends to the same vector for all sybils, the expected pairwise similarity of these updates will increase as the learning process continues. As long as the resulting similarity, including the effect of pardoning between sybils, is below β_m , FoolsGold will adapt the learning rate to 0, satisfying (cond1).

β_m is the point at which the logit function is below 0 and is a function of the confidence parameter κ :

$$\beta_m \geq 1 - \frac{e^{-0.5\kappa}}{1 + e^{-0.5\kappa}}$$

Condition 2: Regarding the ideal gradients of honest clients, we assume that the maximum pairwise cosine similarity between the ideal gradient of honest clients is β_g . As long as β_g is sufficiently low such that FoolsGold assigns a learning rate adaptation of 1 for all honest clients, the second condition of $h(i, t)$ is met. β_g is the point at which the logit function is greater than 1 and is also a function of the confidence parameter κ :

$$\beta_g \leq 1 - \frac{e^{0.5\kappa}}{1 + e^{0.5\kappa}}$$

If the above condition holds, FoolsGold will classify these clients to be honest and will not modify their learning rates. This maintains the constant learning rate and satisfies (cond2).

GhostImage: Remote Perception Attacks against Camera-based Image Classification Systems

Yanmao Man¹, Ming Li¹, and Ryan Gerdes²

¹University of Arizona

²Virginia Tech

Abstract

In vision-based object classification systems imaging sensors perceive the environment and then objects are detected and classified for decision-making purposes; e.g., to maneuver an automated vehicle around an obstacle or to raise an alarm to indicate the presence of an intruder in surveillance settings. In this work we demonstrate how the perception domain can be remotely and unobtrusively exploited to enable an attacker to create spurious objects or alter an existing object. An automated system relying on a detection/classification framework subject to our attack could be made to undertake actions with catastrophic results due to attacker-induced misperception.

We focus on camera-based systems and show that it is possible to remotely project adversarial patterns into camera systems by exploiting two common effects in optical imaging systems, viz., lens flare/ghost effects and auto-exposure control. To improve the robustness of the attack to channel effects, we generate optimal patterns by integrating adversarial machine learning techniques with a trained end-to-end channel model. We experimentally demonstrate our attacks using a low-cost projector, on three different image datasets, in indoor and outdoor environments, and with three different cameras. Experimental results show that, depending on the projector-camera distance, attack success rates can reach as high as 100% and under targeted conditions.

1 Introduction

Object detection and classification have been widely adopted in autonomous systems, such as automated vehicles [1, 2] and unmanned aerial vehicles [3], as well as surveillance systems, e.g., smart home monitoring systems [4, 5]. These systems first perceive the surrounding environment via sensors (e.g., cameras, LiDARs, and motion sensors) that convert analog signals into digital data, then try to understand the environment using object detectors and classifiers (e.g., recognizing traffic signs or unauthorized persons), and finally make a decision on how to interact with the environment (e.g., a vehicle may decelerate or a surveillance system raises an alarm).



(a) Projector off

(b) Projector on

Figure 1: A STOP sign image was injected into a camera by a projector, which was detected by YOLOv3 [17].

While the cyber (digital) attack surface of such systems have been widely studied [6, 7], vulnerabilities in the perception domain are less well-known, despite perception being the first and critical step in the decision-making pipeline. That is, if sensors can be compromised then false data can be injected and the decision making process will indubitably be harmed as the system is not acting on an accurate view of its environment. Recent work has demonstrated false data injection against sensors in a remote manner via either electromagnetic (radio frequency) interference [8], laser pulses (against microphones [9], or LiDARs [10–12]), and acoustic waves [13, 14]. These perception domain sensor attacks alter the data at the source, hence bypassing traditional digital defenses (such as crypto-based authentication or access control), and are subsequently much harder to defend against [15, 16]. These attacks can also be remote in that the attacker needn't physically contact/access/modify devices or objects.

Among the aforementioned sensors, at least for automated systems in the transportation and surveillance domains, cameras are more common/crucial. Existing *remote* attacks against cameras are limited to, essentially, denial-of-service attacks [11, 18, 19], which are easily detectable (e.g., by tampering detection [20]) and for which effective mitigation strategies exist (e.g., by sensor fusion [21]). In this work, we consider attacks that cause camera-based image classification

system to either misperceive actual objects or perceive non-existent objects by remotely injecting light-based interference into a camera, without blinding it. Formally, we consider *creation attacks* whereby a spurious object (e.g., a non-existent traffic sign, or obstacle) is seen to exist in the environment by a camera, and *alteration attacks*, in which an existing object in the camera view is changed into another attacker-determined object (e.g., changing a STOP sign to a YIELD sign or changing an intruder into a bicycle).

As it is not possible, due to optical principles, to directly project an image into a camera, we propose to exploit two common effects in optical imaging systems, viz., *lens flare effects* and *exposure control* to induce camera-based misperception. The former effect is due to the imperfection of lenses, which causes light beams to be refracted and reflected multiple times resulting in polygon-shape artifacts (a.k.a., *ghosts*) to appear in images [22]. Since ghosts and their light sources typically appear at different locations, an attacker can overlap specially crafted ghosts with the target object's without having the light source blocking it. Auto exposure control is a feature common to cameras that determines the amount of light incident on the imager and is used, for example, to make images look more natural. An attacker can leverage exposure control to make the background of an image darker and the ghosts brighter, so as to make the ghosts more prominent (i.e., noticeable to the detector/classifier) and thus increase attack success rates. Fig. 1 presents an example of a creation attack, where we used a projector to inject an image of a STOP sign in a ghost, which is detected and classified as a STOP sign by YOLOv3 [17], a state-of-the-art object detector.

Theoretically arbitrary patterns can be injected via ghosts. However, it is challenging to practically and precisely control the ghosts, in terms of their resolutions and positions in images, making arbitrary injection impracticable in some scenarios. Hence, we propose an empirical projector-camera channel model that predicts the resolution and color of injected ghost patterns, as well as the location of ghosts, for a given projector-camera arrangement. Experimental results show that at short distances attack success rates are as high as 100%, but at longer distances the rates decrease sharply; this is because at long distances ghost resolutions are low, resulting in patterns that cannot be recognized by the classifier.

To improve the efficacy of our attack, which we dub GhostImage, especially at lower resolutions, we assume that the attacker possesses knowledge about the image classification/detection algorithm. Based on this knowledge the attacker is able to formulate and solve an optimization problem to find optimal attack patterns, of varying resolutions, to project that will be recognized by the image classifier as the intended target class [23, 24]; i.e., the pattern projected will yield a classification result of the attacker's choice. As the channel may distort the injected image (in terms of color, brightness, and noise), we extend our projector-camera model to include auto exposure control and color calibration and in-

tegrate the channel model into our optimization formulation. This results in a pattern generation approach that is resistant to channel effects and thus able to defeat a classifier under realistic conditions.

We use self-driving and surveillance systems as two illustrative examples to demonstrate the potential impact of GhostImage attacks. Proof-of-concept experiments were conducted with different cameras, image datasets, and environmental conditions. Results show that our attacks are able to achieve attack success rates as high as 100%, depending on the projector-camera distance. Our contributions are summarized as follows.

- We are the first to study remote perception attacks against camera-based classification systems, whereby the attacker induces misclassification of objects by injecting light, conveying adversarially generated patterns, into the camera.
- Our attack leverages optical effects/techniques, namely, lens flare and auto-exposure control, that are widespread and common, making the attack likely to be effective against most cameras. Furthermore, we incorporate these effects in an end-to-end manner into an adversarial machine learning-based optimization framework to find the optimal patterns an attacker should inject to cause misperception.
- We demonstrate the efficacy of the attacks through experiments with varying image datasets, cameras, distances, and indoor to outdoor environments. Results show that GhostImage attacks are able to achieve attack success rates as high as 100%, depending on the projector-camera distance.

2 System and Threat Model

System and attack models are described, including two attack objectives and the attacker's capabilities.

2.1 System Model

We assume an end-to-end camera-based object classification system (Fig. 2) in which a camera captures an image of a scene with objects of interest. The image is then fed to an object detector to crop out the areas of objects, and finally these areas are given to a neural network to classify the objects. Autonomous systems increasingly rely on such classification systems to make decisions and actions. If the classification result is incorrect (e.g., modified by an adversary), wrong actions could be taken. For example, in a surveillance system, if an intruder is not detected, the house may be broken-in without raising an alarm.

2.2 Threat Model

We consider two different attack objectives. In **creation attacks** the goal is to inject a spurious (i.e., non-existent) object

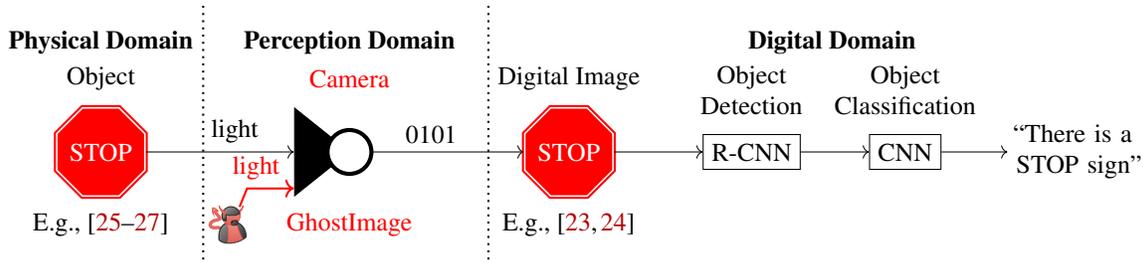


Figure 2: Camera-based object classification systems. GhostImage attacks target the perception domain, i.e., the camera.

into the scene and have it be recognized (classified) as though it were physically present. For **alteration attacks** an attacker injects adversarial patterns over an object of interest in the scene that causes the object to be misclassified.

There are two types of attackers with differing capabilities: **Camera-aware attackers** who possess knowledge of the victim’s camera (i.e., they do not know the configuration of the lens system, nor post-processing algorithms, but they can possess the same type of camera used in the target system), from which they can train a channel model using the camera as a black-box. With such capabilities, they are able to achieve creation attacks and alteration attacks. **System-aware attackers** not only possess the capabilities of the camera-aware attackers, but also know about the image classifier including its architecture and parameters, i.e., black-box attacks on the camera but white-box attacks on the classifier. With such capabilities, it is able to achieve creation attacks and alteration attacks as well, but with higher attack success rates.

Both types of attackers are remote (unlike the lens sticker attack [28]), i.e., they do not have access to the hardware or the firmware of the victim camera, nor to the images that the camera captures. We assume that both attackers are able to track and aim victim cameras [12, 18, 29].

3 Background

In this section, we will introduce optical imaging principles, including flare/ghost effects and exposure control, which we will exploit to *realize* GhostImage attacks. Then, we will discuss the preliminaries about neural networks and adversarial examples that we will use to *enhance* GhostImage attacks.

3.1 Optical Imaging Principles

Due to the optical principles of camera-based imaging systems, it is not feasible to directly point a projector at a camera, hoping that the projected patterns can appear at the same location with the image of the targeted object, because the projector has to obscure the object in order to make the two images overlap. Instead, we exploit lens flare effects and auto exposure control to inject adversarial patterns.

Lens flare effects [22, 30] refer to a phenomenon where one or more undesirable artifacts appear on an image because bright light get scattered or flared in a non-ideal lens system (Fig. 3). Ideally, all light beams should pass directly through the lens and reach the CMOS sensor. However, due to the quality of the lens elements, a small portion of light gets reflected several times within the lens system and then reaches the sensor, forming multiple polygons (called “ghosts”) on the image. The shape of polygons depends on the shape of the aperture. For example, if the aperture has six sides, there will be hexagon-shaped ghosts in the image. Normally ghosts are very weak and one cannot see them, but when a strong light source (such as the sun, a light bulb, a laser, or a projector) is present (unnecessarily captured by the CMOS sensor, though [31]), the ghost effects become visible. Fig. 3 shows only one reflection path, but there are many other paths and that is why there are usually multiple ghosts in an image.

Existing literature [22] about ghosts focused on the simulation of ghosts given the detailed lens configurations, in which the algorithms simulate every possible reflection path. Such white-box models are computationally expensive, and also requires white-box knowledge of internal lens configurations, thus are not suitable for our purposes. In Sections 4 and 5, we study flare effects in a black-box manner (more general than Vitoria et al. [30]), where we train a lightweight end-to-end model that is able to predict the locations of ghosts, estimate the resolutions within ghost areas, and also calibrate colors.

Exposure control mechanisms [32] are often equipped in cameras to adjust brightness by changing the size of the aperture or the exposure time. In this work, we will model and exploit auto exposure control to manipulate the brightness

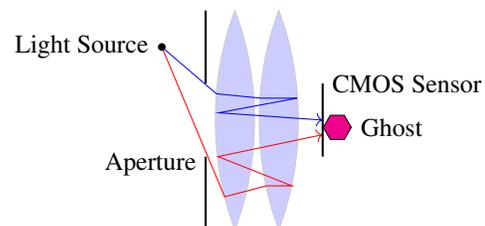


Figure 3: Ghost effect principle

balance between the targeted object and the injected attack patterns in ghosts.

3.2 Neural Nets and Adversarial Examples

We abstract a neural network as a function $Y = f_{\theta}(x)$ and we omit the details of it due to the page limit. The input $x \in \mathbb{R}^{w \times h \times 3}$ (width, height and RGB channels) is an image, $Y \in \mathbb{R}^m$ is the output vector, and θ is the parameters of the network (which is fixed thus we omit it for convenience). A softmax layer is usually added to the end of a neural network to make sure that $\sum_{i=1}^m Y_i = 1$ and $Y_i \in [0, 1]$. The classification result is $C(x) = \text{argmax}_i Y_i$. Also, the inputs to the softmax layer are called *logits* and denoted as $Z(x)$.

An adversarial example [23] is denoted as y , where $y = x + \Delta$. Here, Δ is additive noise that has the same dimensionality with x . Given a benign image x and a target label t , an adversary wants to find a Δ such that $C(x + \Delta) = t$, i.e., *targeted attacks*. Note that, in this paper, the magnitude of Δ is not constrained below a small threshold, since the perceived images are usually not directly observed by human users. But we still try to minimize it because it represents the attack power and cost.

4 Camera-aware GhostImage Attacks

In this section, we will discuss how a camera-aware attacker is able to inject arbitrary patterns in the perceived image of the victim camera using projectors.

4.1 Technical Challenges

Since we assume that the attacker do not have access to the images that the targeted camera captures, he/she will have to be able to predict how ghosts might appear in the image. First, the locations of ghosts should be predicted given the relevant positions of the projector and the camera, so that the attacker can align the ghost with the image of the object of interest to achieve alteration attacks. Second, since a projector can inject shapes in ghost areas, the attacker needs to find out the maximum resolution of shapes that it can inject. Lastly, it is also challenging to realize the attacks derived from the position and resolution models above with a limited budget.

4.2 Ghost Pixel Coordinates

Given the pixel coordinates of the target object G (Fig. 4a), we need to derive the real-world coordinates A' of the projector so that we know where to place the projector in order to let one of the ghosts overlap with the image of the object. To do this, we derive the relationship between G and A' in two steps: We first calculate the pixel coordinates of the light source A given A' , and then we calculate G based on A .

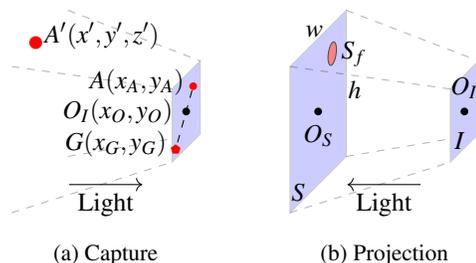


Figure 4: Capture and projection are reverses of each other.

Based on homogeneous coordinates [33], assuming the camera is at the origin of the coordinate system, we have

$$(u, v, w)^{\top} = M_c \cdot (x', y', z', 1)^{\top}, \quad (1)$$

where M_c is the camera's geometric model [33], a 3×4 matrix. M_c can be trained from another (similar) camera, and then be applied to the victim camera. The coordinates of A is then $A = (x_A, y_A)^{\top} = (u/w, v/w)^{\top}$, by the homogeneous transformation. Note that, A does not have to appear in the view of the camera, which makes the attack more stealthy [31].

In order to find the relationship of the pixel coordinates between light sources A and their ghosts G , we did a simple experiment where we moved around a flashlight in front of the camera [34], and recorded the pixel coordinates of the flashlight and the ghosts. Similar to Vitoria et al.'s results [30], we observe that, for each G , we have $\overline{AO_I/O_I G} = r_G$ (being constant), wherever A is, and $r \in (-\infty, \infty)$. This means the feasible region for the placement of the projector is large; to attack an autonomous vehicle, for example, it can be located on an overbridge, on a traffic island, or even in the preceding vehicle or on a drone, etc. Finally, given $A = (x_A, y_A)$, $O_I = (x_O, y_O)$ and r , we can derive the coordinates of ghosts as

$$G = \begin{pmatrix} x_O - (x_A - x_O)/r \\ y_O - (y_A - y_O)/r \end{pmatrix}. \quad (2)$$

With G 's coordinates, the attacker is able to predict the pixel location of ghosts and try adjusting the position and orientation (which implies the angle) of the light source in the real world so as to align one or more ghosts with the image of the object, whose pixel coordinates can be derived using (1), too.

4.3 Ghost Resolution

In our daily life, ghosts normally appear as pieces of single-color polygon-shaped artifacts; this is because the light sources that cause these regular ghosts are single-point sources of light that have just one single color, such as light bulbs, flashlights, etc. In this work, however, we find out that one is able to bring patterns into these ghost areas by simply using a low-cost projector, a special source of light that shines variant patterns in variant colors. For example, in Fig. 1, an

image of a STOP sign that is projected by a projector, appears in one of the ghost areas in the image; this is because the pixel resolution of the projector is high enough that multiple light beams in different colors (got reflected among lenses and then) go into the same ghost. In this subsection, we study the resolution of the patterns in ghost areas¹.

Let us first define the *throwing ratio* of a projector. In Fig. 4b, let plane S be the projected screen (e.g. on a wall), whose height and width are denoted as h and w , respectively. The distance $d = \overline{O_S O_I}$ is called the throwing distance. The throwing ratio of this projection is $r_{\text{throw}} = d/w$. The (physical) size of the projected screen at the victim camera's location is denoted S_O , a part of which is captured by the CMOS sensor of the camera in the ghost area, and we denote the (physical) size of that area as S_f . Let us also define the resolution of the entire projected screen as P_O in terms of pixels (e.g., 1024×768), and the resolution of the ghost as P_f . Clearly, there is a linear relationship among them: $P_f/P_O = S_f/S_O$, where $S_O = wh$. Finally, we can calculate the resolution of the ghost given d and r_{throw} :

$$P_f = \frac{P_O S_f}{\frac{h}{w} \left(\frac{d}{r_{\text{throw}}} \right)^2}. \quad (3)$$

Here, S_f is a constant because the size of the lens is fixed; e.g., the camera [34] has $S_f = 0.0156 \text{ cm}^2$.

4.4 Attack Realization and Experiment Setup

According to Eq. 3, if the attacker wants to carry out long-distance and high-resolution GhostImage attacks, he/she needs a projector with a large throwing ratio r_{throw} . However, the factory longest-throw lenses (NEC NP05ZL Zoom Lens [37]) of our projector can achieve a throwing ratio of maximum 7.2 (which means 9×9 at one meter), and expensive (about \$1600). Instead, we use a cheap (\$80) zoom lens (Fig. 5, Right) [36] that was originally designed for Canon cameras. In our experiments, such a configuration is interestingly feasible² (Fig. 5), achieving the maximum throwing ratio of 20 when the focal length is 250 mm, which means that at a distance of one meter, 32×32 -resolution attacks can be achieved. See Sec. 7.1 for more discussion on lens and projector selection.

Fig. 5 (left) shows a general diagram of GhostImage attacks, where the light source (i.e., a projector) is pointing at the camera from the side, so that the camera can still capture the object (e.g., a STOP sign) for alteration attacks. The light source injects light interference (marked in blue) into the camera, which gets reflected among the lenses of the camera, resulting in ghosts that overlap with the object in the image.

¹We are interested in the resolution of the projector pixels, not camera pixels; a projector pixel is usually captured by multiple camera pixels.

²Because projectors and cameras are dual devices (Fig. 4), their lenses are interchangeable.

Accordingly, a photo of our in-lab experiment setup is given in Fig. 5. The Canon lens was loaded in the NEC projector, though it cannot be seen in the photo. We will evaluate our attack on three different cameras (Sec. 6.2.3).

To mount a creation attack, the attacker computes the maximum resolution P_f for the ghost with a distance d based on (3), and then *downsamples* the target image to the resolution P_f in order to fit in the ghost area. The attacker chooses downsampling as a heuristic approach because he/she is not aware of the classification algorithm.

To mount alteration attacks, in addition to (3) for downsampling, the attacker also needs to consider the pixel coordinates (Eq. 2) of the ghost because the attacker needs to align the ghost with the image of the object of interest so that the resulting, combined image deceives the classifier.

4.5 Camera-aware Attack Evaluation

We substantiate camera-aware attacks on an image classification system that we envision would be used for automated vehicles. Specifically, images, taken by an Aptina MT9M034 camera [34], are fed to a traffic sign image classifier trained on the LISA dataset [38]. In Sec. 6, we will evaluate classification systems for other applications, with different cameras and different datasets.

4.5.1 Dataset and neural network architecture

In order to train an unbiased classifier, we selected eight traffic signs (with 80 instances) from the LISA dataset [38] (Fig. 12a). The network architecture is identical to [25]. We used 80% of samples from the balanced dataset to train the network and the rest 20% to test the network; it achieved an accuracy of 96%.

4.5.2 Evaluation methodology

We iterated five distances, m source classes, m target classes. For each target class, we sampled k images randomly from the dataset. For every combination, we first downsampled the target image based on (3), and projected the image at the camera using the NEC projector. We then took the captured image, cropped out the ghost area, and used the classifier to classify it. If the classification result is the target class, we count it as a successful attack. The procedure for creation attacks is slightly different: Rather than printed traffic signs, we placed a blackboard as the background as it helped us locate the ghosts. Given a throwing ratio of 20 (thanks to the Canon lens) we evaluated five different distances from one meter to five meters. Based on (3), they resulted in 32×32 , 16×16 , 8×8 , 4×4 , and 2×2 resolutions, respectively.



Figure 5: (Left) Attack setup diagram. (Middle) In-lab experiment setup. (Right) Attack equipments: We replaced the original lens of the NEC NP3150 Projector [35] with a Canon EFS 55-250 mm zoom lens [36].

4.5.3 Results

The results about attack success rates of camera-aware attacks at varying distances are shown in Table 1 (Fig. 6 illustrates two successful camera-aware attacks). For the digital domain, we simply added attack images Δ on benign images x as $y = (x + \Delta) / \|x + \Delta\|_\infty$. Based on these experiments, we observe: First, as the distance increases, the success rate decreases. This is because lower-resolution images are less well recognized by the classifier. Second, digital domain results are better than perception domain one, because images are distorted by the projector-camera channel effects. Third, creation attacks result in higher success rates than alteration attacks do because in alteration attacks there are benign images in the background, encouraging the classifier to make correct classifications. We will address these issues in the next section, so as to increase the overall attack success rate.

5 System-aware GhostImage Attacks

There are some limitations of the camera-aware attack introduced in the previous section. First, increasing distances results in lower success rates because the classifier cannot recognize the resulting low-resolution images. Second, there are large gaps between digital domain results and perception domain results, as channel effects (which cause the inconsistency between the intended pixels and the perceived pixels) are not taken into account. In this section, we resolve these limitations and improve GhostImage attacks' success rates by proposing a framework which consists of a channel model that predicts the pixels perceived by the camera, given the pixels as input to the projector, as well as an optimization

formulation based on which the attacker can solve for optimal attack patterns that cause misclassification by the target classifier with high confidence.

5.1 Technical Challenges

First, the injected pixel values are often difficult to control as they exhibit randomness due to variability of the channel between the projector and the camera, thus the adversary is not able to manipulate each pixel deterministically. Second, to achieve optimal results, the attacker needs to precisely predict the projected and perceived pixels, thus channel effects must be modeled in an end-to-end manner, i.e., considering not only the physical channel (air propagation), but also the internal processes of the projector and the camera. Lastly, the resolution of attack patterns is limited by distances and projector lens (Eq. 3), thus the ghost patterns must be carefully designed to fit the resolution with few degrees of freedom.

5.2 System-aware Attack Overview

The system-aware attacker aims to find optimal patterns that can cause misclassification by the target classifier with high confidence by taking advantage of the non-robustness of the classifier [23]. We adopt an adversarial example-based optimization formulation into GhostImage attacks, in which the attacker tries to solve

$$\Delta^* = \arg \min_{\Delta} \|\Delta\|_p + c \cdot \mathcal{L}_{\text{adv}}(y, t, \theta), \quad (4)$$

where Δ is the digital attack pattern as input to the projector, y is the perceived image of the object of interest under attacks, t

Table 1: Camera-aware attack success rates

Distances (meter)	Creation Attacks		Alteration Attacks	
	Digital	Perception	Digital	Perception
1	98%	41%	95%	33%
2	98%	36%	88%	33%
3	80%	34%	67%	34%
4	36%	15%	28%	10%
5	14%	10%	13%	0%



Figure 6: Camera-aware attack examples at one meter in perception domain. Left: Creating a Merge sign. Right: Altering a STOP sign (in the background) into a Merge sign.

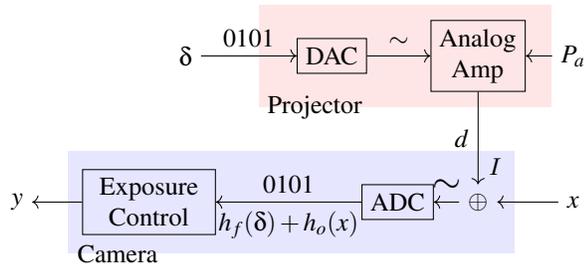


Figure 7: Projector-camera channel model

is the target class, and θ represents the targeted neural network. $\|\cdot\|_p$ is an ℓ_p -norm that measures the magnitude of a vector, and \mathcal{L}_{adv} is a loss function indicating how (un)successful Δ is. Here, we aim to minimize the power of the projector required for a successful attack, meanwhile maximizing the successful chance of attacks. The relative importance of these two objectives is balanced by a constant c . Sec. 5.4 details (4) in terms of how we handle Δ being a non-negative tensor that is also able to depict grid-style patterns in different resolutions.

More importantly, in (4) y is the final perceived image used as input to the classifier, which is estimated by our channel model in an end-to-end style (Fig. 7), in which δ^3 is the input to the projector, and y is the resulting image captured by the camera. The model can be formulated as

$$y = g(h_f(\Delta) + h_o(x)). \quad (5)$$

where $h_f(\Delta)$ is the ghost model that estimates the perceived adversarial pixel values in the ghost. For simplicity we let $h_o(x) = x$ because the attacker possesses same type of the camera so that x can be obtained a priori, and $g(\cdot)$ is the auto exposure control that adjusts the brightness. Sec. 5.3 introduces the derivation of (5).

Next, we will first present the channel model, and then formulate the optimization problem for finding the optimal adversarial ghost patterns.

5.3 Projector-Camera Channel Model

We consider the projector to camera channel model (Fig. 7) in which δ is an RGB value the attacker wishes to project which is later converted to an analog color by the projector. The attacker can control the power (P_a) of the light source of the projector so that the luminescence can be adjusted. The targeted camera is situated at a distance of d , which captures the light coming from both the projector and reflected off the object (x). The illuminance received by the camera from the projector is denoted as I . The camera converts analog signals into digital ones, based on which it adjusts its exposure, with the final, perceived RGB value being y . An ideal

³Different than Δ which is a $w \times h \times 3$ tensor, δ is a single pixel with dimension 3×1 for the convenience of the analysis.

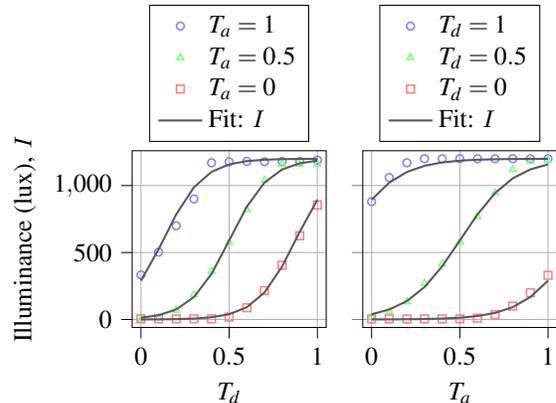


Figure 8: Illuminance depends on the RGB amplitude T_d , and the light bulb intensity T_a .

channel would yield $y = x + \delta$ but due to channel effects, we need to find a way to adjust the projected RGB value such that the perceived RGB value is as intended, i.e., to find the appropriate x given y .

5.3.1 Exposure control

As we discussed in Section 3.1, cameras are usually equipped with auto-exposure control, where according to the overall brightness of the image, the camera adjusts its exposure by changing the exposure time, or the size of its aperture, or both. We observed from our experiments that, as we increase the luminescence of the projector (I), in the image the brightness of the object (x) decreases but the ghost (δ) does not decrease as much. Modeling such phenomena helps the attacker to precisely predict the perceived image. For the following, we will first find out how the illuminance I depends on δ and P_a (the normalized power of light bulb ranging from 0% to 100%), and then analyze how y depends on I .

How does I depend on δ and P_a ? We conducted a series of experiments, where $T_d = \|\delta\|_\infty = \max_i \delta_i$ and P_a were varied. We recorded the illuminance directly in front of the camera using an illuminance meter, with the projector one meter away. The results are plotted in Fig. 8, which shows that

$$I(T_d, P_a, d) = \frac{c_d}{d^2} \cdot \frac{I_{\max}}{1 + e^{-t}}, \quad (6)$$

where $t = a \times T_d + b \times P_a + c_t$, and a, b, c_d and c_t are constants derived from the data. I_{\max} is the maximum illuminance of the projector at a distance of one meter. Such a sigmoid-like function captures the luminescence saturation property of the projector hardware.

How does the perceived x depend on I ? In the same experiments we also recorded the RGB value of the ghost (δ)

with a blackboard as background (in order to reduce ambient impacts), and a piece of white paper (x) that was also on the blackboard but did not overlay with the ghost. Their data are shown in Fig. 9, from which we can derive the *dimming ratio* that measures the change of exposure/brightness:

$$\gamma(I) = \frac{I_{\text{env}}}{I + I_{\text{env}}}, \quad (7)$$

where I_{env} is the ambient lighting condition in illuminance which differs from indoors to outdoors for instances. From this equation, we see that in an environment with static lighting condition, as the luminescence of the projector increases, the dimming ratio decreases, hence the objects become darker. With (7), the adversary is able to conduct real-time attacks by simply plugging in the momentary I_{env} .

How does the perceived δ depend on I ? When $x = 0$, $\|y_f\| = \|y_f\|_{\infty}$ (the lower subplot of Fig. 9) depends on I in two ways:

$$\|y_f\|(I) = \gamma(I) \cdot \rho \cdot I.$$

On one hand, the last term I increases the intensity of ghosts, but on the other hand the dimming ratio $\gamma(I)$ dims down ghost, whereby ρ is a trainable constant. With this, we can rewrite the perceived flare as

$$y_f = \|y_f\| H_c \frac{\delta}{\|\delta\|},$$

where H_c is the color calibration matrix to deal with color distortion, which will be discussed in Section 5.3.2. The term $1/\|\delta\|$ normalizes δ . In the end, we have the channel model

$$y = \gamma(I) \left(\rho I H_c \frac{\delta}{\|\delta\|} + x \right). \quad (8)$$

Compared to (5),

$$h_f(\delta) = \rho I H_c \frac{\delta}{\|\delta\|}, \quad g(t) = \gamma(I)t, \quad h_o(x) = x.$$

With (8), the attacker is able to predict how bright and what colors/pixel values the ghost and the object will be, given the projected pixels, the power of the projector, and the distance.

5.3.2 Color calibration

Considering a dark background (i.e., $x = 0$), (8) can be simplified as $y = \gamma(I) \rho I H_c \delta / \|\delta\|$, where H_c is a 3×3 matrix (as three color channels) that calibrates colors. Both y and δ are 3×1 column vectors. H_c should be an identity matrix for an ideal channel, but due to the color-imperfection of both the projector and the camera, H_c needs to be learned from data. To simplify notations, we define corrected x and y as

$$\hat{x} = \frac{\delta}{\|\delta\|}, \quad \hat{y} = \frac{y}{\rho I \gamma(I)},$$

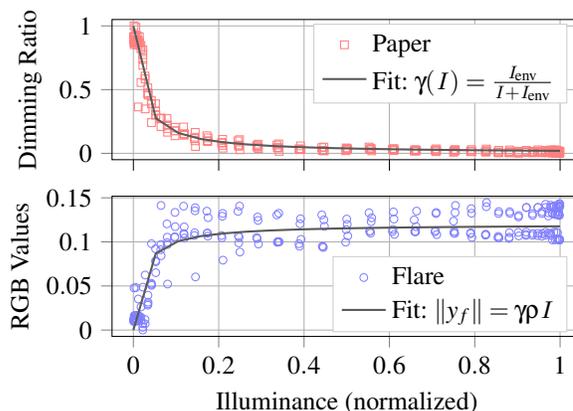


Figure 9: Perceived RGB values v.s. illuminance.

so that we can write

$$\hat{y} = H_c \hat{x}.$$

We did another set of experiments where we collected $n = 100$ pairs of (\hat{x}, \hat{y}) with dark background (to make $x = 0$), with δ being assigned randomly, and $P_a = 30\%$. We grouped them into X and Y :

$$X = [\hat{x}_1^\top, \hat{x}_2^\top, \dots, \hat{x}_n^\top]^\top, \quad Y = [\hat{y}_1^\top, \hat{y}_2^\top, \dots, \hat{y}_n^\top]^\top,$$

where both X and Y are $n \times 3$ matrices. We solve

$$\min_{H_c} \|Y - X H_c\|_2^2.$$

to obtain H_c , which is known as a non-homogeneous least square problem [33], and has a closed-form solution:

$$H_c = \left((X^\top X)^{-1} X^\top Y \right)^\top.$$

Plugging H_c back to (8) completes our channel model.

5.3.3 Model validation

Fig. 10 demonstrates the accuracy of our channel model. In it the left image is the original input to the projector, the middle image is the estimated output from the camera based on our channel model (Eq. 8), and the image on the right is the actual image in a ghost captured by the camera. As can be seen, the difference between the actual and predicted is much less than the actual and original. While blurring effect is apparent in the actual y , we do not model it but the success rates are still high despite it. As we will see in Section 6, our channel model is general enough that once trained on one camera in one environment, it can be transferred to different environments and different cameras without retraining.

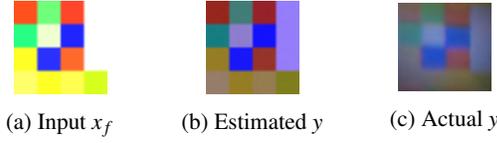


Figure 10: An example of channel model prediction

5.4 Optimal Adversarial Projection Patterns

In long-distance, low-resolution GhostImage attacks there are only a few pixels in the ghost area. A camera-aware attacker’s strategy is to simply downsample attack images into low resolutions, but that does not result in high success rates. While (4) is abstract, for the rest of this subsection, we will progressively detail it and show how it can be solved in light of the channel model to improve attack success rates. We will start with the simplest case where adversarial perturbations are random noise (Sec. 5.1). Then, single-color ghosts will be introduced. Later, we will consider how to find semi-positive additive noise due to the fact that superposition can only increase perceived light intensity but not decrease it. Finally, we examine the optimization problem to find optimal ghost patterns in grids at different resolutions.

5.4.1 Ghosts in random noise

Let us consider the simplest case first where the random noise Δ is drawn from one single Gaussian distribution for all three channels, i.e., $\Delta \sim \mathcal{N}(\mu, \sigma^2)$, where the size of Δ is $w \times h \times 3$ with w and h representing the width and height of the benign image x . This is because the values of each pixel that appear in the ghost area follow Gaussian distributions according to statistics obtained from our experiments.

The adversary needs to find μ and σ such that when Δ is added to the benign image x , the resulting image y will be classified as the target class t . That said, the logits value (Section 3.2) of the target class should be as high as possible compared with the logits values of other classes [24]. Such a difference is measured by the loss function $\mathcal{L}_{\text{adv}}(y, t)$

$$\mathcal{L}_{\text{adv}}(y, t) = \max \left\{ -\kappa, \max_{i: i \neq t} \{ \mathbb{E}[Z_i(y)] \} - \mathbb{E}[Z_t(y)] \right\}, \quad (9)$$

where $\mathbb{E}[Z_i(y)]$ is the expectation of logits values of Class i given the input y . Term $\max_{i: i \neq t} \{ \mathbb{E}[Z_i(y)] \}$ is the highest expected logits value among all the classes except the target class t , while $\mathbb{E}[Z_t(y)]$ is the expected logits value of t . Here, κ controls the logits gap between $\max_{i: i \neq t} \{ \mathbb{E}[Z_i(y)] \}$ and $\mathbb{E}[Z_t(y)]$; the larger the κ is, the more confident that Δ is successful. The attacker needs \mathcal{L}_{adv} as low as possible so that the neural network would classify y as Class t . Most importantly, y is computed based on our channel model (Eq. 8), so that the optimizer finds the optimal ghost patterns that are resistant to the channel effects. Unfortunately, due to the complexity of neural networks, the expectations of logits values

$\mathbb{E}[Z_i(y)]$ are hard to be expressed analytically; we instead use Monte Carlo methods to approximate it:

$$\hat{\mathbb{E}}[Z_i(y)] = \frac{1}{T} \sum_{j=1}^T Z_i(y_j),$$

where T is the number of trials, and y_j is of the j -th trial.

Meanwhile, the adversary also needs to minimize the magnitude of Δ to reduce the attack power and noticeability, as well as its peak energy consumption, quantified by σ . The expectation of the magnitude of Δ is

$$\mathbb{E}[\|\Delta\|_p] = \mu n^{1/p}, \quad \text{with } n = 3wh. \quad (10)$$

Putting (9) and (10) together with a tunable constant c , we have our optimization problem for the simplest case

$$\begin{aligned} \mu^*, \sigma^* &= \underset{\mu, \sigma}{\text{arg min}} \quad \mathbb{E}[\|\Delta\|_p] + \sigma + c \cdot \mathcal{L}_{\text{adv}}(y, t), \\ &\text{subject to} \quad \sigma > \sigma_l, \end{aligned}$$

Here, σ_l is the lower bound of the standard deviation σ , meaning that the interference generator and the channel environment can provide random noise with the standard deviation of at least σ_l . When $\sigma_l = 0$, the adversary is able to manipulate pixels deterministically. Therefore, when we fix σ as σ_l in the optimization problem, the attack success rate when deploying μ^* would be the lower bound of the attack success rate. In other words, the adversary equipped with an attack setup that can produce noise with a lower variance (than σ_l^2) can carry out attacks with higher success rates. Therefore, we can simplify our formulation by removing the constraint about σ , so the optimization problem becomes

$$\mu^* = \underset{\mu}{\text{arg min}} \quad \mathbb{E}[\|\Delta\|_p] + c \cdot \mathcal{L}_{\text{adv}}(y, t). \quad (11)$$

For the rest of the paper we will simply use σ to denote σ_l .

5.4.2 Ghosts in single-color

Since in (11) there is only one variable that the adversary is able to control, it is infeasible to launch a targeted attack with such few degrees of freedom. As a result, the adversary needs to manipulate each channel individually. That is, for each channel, there will be an independent distribution from which noise will be drawn. This is feasible because noise can appear in different colors in the ghost areas in which three channels are perturbed differently when using projectors. Let us decompose Δ as $\Delta = [\Delta_R, \Delta_G, \Delta_B]$, where the dimension of $\Delta_{\{R,G,B\}}$ is $w \times h$, and they follow three independent Gaussian distributions

$$\Delta_R \sim \mathcal{N}(\mu_R, \sigma_R^2), \quad \Delta_G \sim \mathcal{N}(\mu_G, \sigma_G^2), \quad \Delta_B \sim \mathcal{N}(\mu_B, \sigma_B^2).$$

Here, $\mu_{\{R,G,B\}}$ and $\sigma_{\{R,G,B\}}$ are the means and the standard deviations (σ) of the three Gaussian distributions, respectively.

The expectation of such Δ is then

$$\mathbb{E}[\|\Delta\|_p] = \left[\frac{n}{3} (\mu_R^p + \mu_G^p + \mu_B^p) \right]^{\frac{1}{p}}. \quad (12)$$

(10) is a special case of (12) when $\mu = \mu_R = \mu_G = \mu_B$. We denote $\boldsymbol{\mu} = [\mu_R, \mu_G, \mu_B]^\top$. Hence, similar to (11), we have the optimization problem for single-color perturbation [39]

$$\boldsymbol{\mu}^* = \arg \min_{\boldsymbol{\mu}} \mathbb{E}[\|\Delta\|_p] + c \cdot \mathcal{L}_{\text{adv}}(y, t), \quad (13)$$

by which the adversary finds $\boldsymbol{\mu}^*$ from which Δ is drawn.

5.4.3 Ghost grids

Since projector’s pixels are arranged in grids, the attack patterns are in grids as well, especially in lower resolutions. We enable Δ with patterns in different resolutions. Such a grid pattern $\mathbf{\Delta}$ can be composed of several blocks $\Delta_{i,j,k}$, i.e., $\Delta_{i,j,k} : \{1 \leq i \leq N_{\text{row}}, 1 \leq j \leq N_{\text{col}}, 1 \leq k \leq N_{\text{chn}}\}$ where N_{row} , N_{col} and N_{chn} is the number of rows, columns, and channels of a grid pattern, respectively, in terms of blocks. In a word, $\Delta_{i,j,k}$ is the perturbation block at i -th row, j -th column and k -th channel. A block $\Delta_{i,j,k}$ is a random matrix and its size is $\frac{w}{N_{\text{col}}} \times \frac{h}{N_{\text{row}}}$, so that the size of $\mathbf{\Delta}$ is still $w \times h \times 3$. Besides, the elements in the random matrix $\Delta_{i,j,k}$ is i.i.d. drawn from a Gaussian distribution, i.e., $\Delta_{i,j,k} \sim \mathcal{N}(\mu_{i,j,k}, \sigma^2)$.

The adversary finds the optimal grid pattern $\mathbf{\Delta}$ by solving

$$M^* = \arg \min_M \mathbb{E}[R(\|\mathbf{\Delta}\|_p)] + c \cdot \mathcal{L}_{\text{adv}}(y, t), \quad (14)$$

where $R(\mathbf{\Delta})$ is the softplus function to guarantee that the perturbation is always positive. $M = \{\mu_{i,j,k}\}$ is a tensor in shape $N_{\text{row}} \times N_{\text{col}} \times N_{\text{chn}}$. See Fig. 12a for some examples of adversarial grids in different resolutions.

6 System-aware Attack Evaluation

In this section, we consider camera-based image classification systems, as used in self-driving vehicles and surveillance systems, to illustrate the potential impact of our attacks. We present proof-of-concept system-aware attacks in terms of *attack effectiveness*, namely how well system-aware attacks perform in the same setup as camera-aware attacks (Section 4.5), and *attack robustness*, namely how well system-aware attacks are when being evaluated in different setups.

We will again use attack success rates as our metric. We used the Adam Optimizer [40] to solve our optimization problems. There are two sets of results: *Emulation results* refer to the classification results on emulated, combined images of benign images and attack patterns using our channel model (Equation 8). Emulation helps us conduct scalable and fast

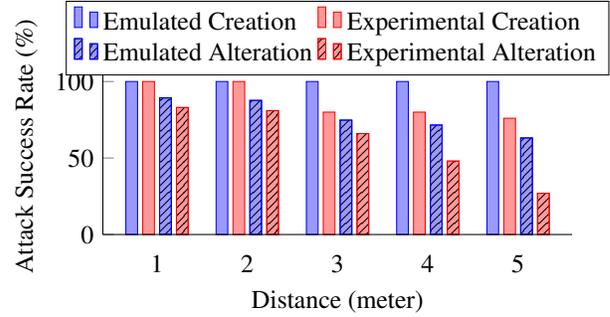


Figure 11: System-aware creation and alteration

evaluations of GhostImage attacks before conducting real-world experiments⁴. *Experimental results* refer to the classification results on the images that are actually captured by the victim cameras when the projector is on.

6.1 Attack Effectiveness

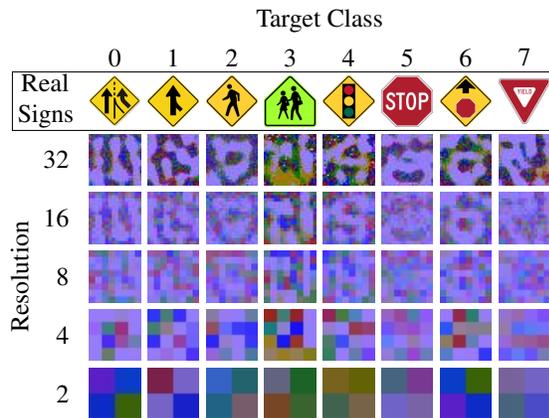
To compare with camera-aware attacks, system-aware attacks are evaluated in a similar procedure, targeting a camera-based object classification system with the LISA dataset and its classifier. The system uses an Aptina MT9M034 camera [34] in an in-lab environment.

6.1.1 Creation attacks

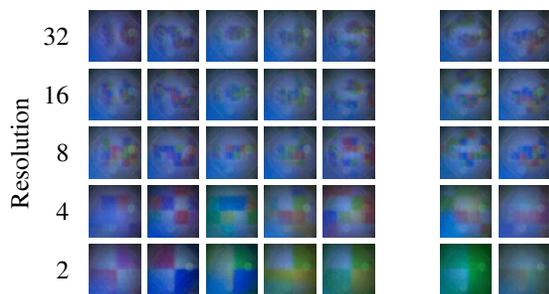
For emulated creation attacks, all distances (or all resolutions) yield attack success rates of 100% (Fig. 11), which means that our optimization problem is easy to solve. In terms of computational overhead, we need roughly 30s per image at 2×2 -resolution, and 10s at 4×4 or above (because of more degrees of freedom) using an NVIDIA Tesla P100 [41]. Fig. 12a shows examples of emulated attack patterns for creation attacks, along with the images of real signs on the top. Interestingly, high-resolution shapes do look like real signs. For example, we can see two vertical bars for ADDEDLANE, and also we can see a circle at the middle south for STOPAHEAD, etc. These results are consistent with the ones from the MNIST dataset [42] where we could also roughly observe the shapes of digits. Secondly, they are blue tinted because our channel model suggests that ghosts tend to be blue, thus the optimizer is trying to find “blue” attack patterns that are able to deceive the classifier.

Interestingly, the all k resulting patterns of solving the optimization problem targeting one class from k different (random) starting points look similar to the ones shown in Fig. 12a. However, CIFAR-10 [43] and ImageNet [44] yield much different results: those patterns look rather random compared to the results from LISA or MNIST. The reason might be that in CIFAR-10, images in the same category are still very

⁴Source code is at <https://github.com/harry1993/ghostimage>



(a) Emulated creation attacks



(b) Experimental alteration attacks

Figure 12: System-aware attack pattern examples.

different, such as two different cats, but in LISA, two images of STOP signs do not look as different as two cats.

For the experimental results of creation attacks, we see that as distances increase, success rates decrease a little (Fig. 11), but much better than the camera-aware attacks (Table 1), because the optimization formulation helped find those optimal attack patterns with high confidence.

6.1.2 Alteration attacks

The emulated and experimental results of alteration attacks are shown in Fig. 11. Compared with creation attacks, alteration attacks perform a bit worse, especially for large distances (three meters or further). This is because the classifier also “sees” the benign image in the background and tends to classify the entire image as the benign class. Moreover, the alignment of attack patterns and the benign signs is imperfect. However, when we compare Fig. 11 with Table 1 for camera-aware alteration attacks, we can see large improvements. Fig. 12b provides an example of system-aware alteration attacks in the perception domain, which were trying to alter the (printed) STOP sign into other signs: they look “blue” as the channel model predicted. The fifth column is not showing as it is STOP.

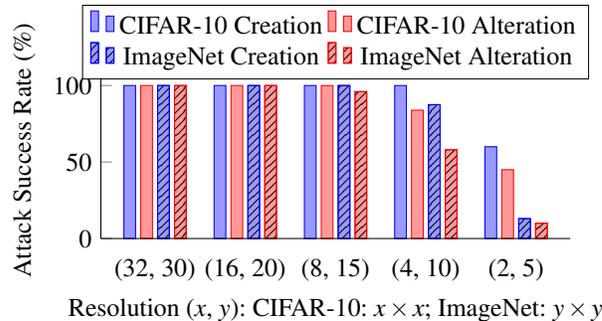


Figure 13: System-aware attacks on CIFAR-10 and ImageNet

6.2 Attack Robustness

We evaluate the robustness of our attacks in terms of different datasets, environments, and cameras.

6.2.1 Different image datasets

Here we evaluate our system-aware attacks on two other datasets, CIFAR-10 [43] and ImageNet [44], by emulation only because previous results show that our attack emulation yields similar success rates as experimental results.

CIFAR-10 The network architecture and model hyper parameters are identical to [24]. The network was trained with the distillation defense [45] so that we can evaluate the robustness of our attacks in terms of adversarial defenses. A classification accuracy of 80% was achieved. The evaluation procedure is similar to Sec. 4.5.2. Results are shown in Fig. 13. The overall trend is similar to the LISA dataset, but the success rates are significantly higher. The reason might still be the large variation within one class (Section 6.1.1), so that the CIFAR-10 classifier is not as sure about one class as the LISA classifier is, hence is more vulnerable to GhostImage attacks.

ImageNet We used a pre-trained Inception V3 neural network [46] for the ImageNet dataset to evaluate the attack robustness against large networks. Since the pre-trained network can recognize 1000 classes, we did not iterate all of them [24]. Instead, for alteration attacks, we randomly picked ten benign images from the validation set, and twenty random target classes, while for creation attacks, the “benign” images were purely black. Results are given in Fig. 13.

For high resolutions ($\geq 15 \times 15$), the attack success rates were nearly 100%. But as soon as the resolutions went down to 10×10 or below, the rates decreased sharply. The reason might be that in order to mount successful *targeted* attacks on a 1000-class image classifier, a large number of degrees of freedom are required. 10×10 or lower resolutions plus three color channels might not be enough to accomplish targeted attacks. To verify this, we also evaluated untargeted alteration



Figure 14: Outdoor experiment setup

attacks on ImageNet. Results show that when the resolutions are 1×1 or 2×2 , the success rates are 50% or 80%, respectively. But as soon as the resolutions go to 3×3 or above, the success rates reach 100%. Lastly, similar to CIFAR-10, system-aware attacks on ImageNet were more successful than on LISA, because of the high variation within one class.

6.2.2 Outdoor experiments

In order to evaluate system-aware attacks in a real-world environment, we also conducted experiments outdoor (Fig. 14), where the camera was put on the hood of a vehicle that was about to pass an intersection with a STOP sign. The attacker’s projector was placed on the right curb, and it was about four meters away from the camera. The experiments were done at noon, at dusk and at night (with the vehicle’s front lights on) to examine the effects of ambient light on attack efficacy. The illuminances were 4×10^4 lx, 4×10^3 lx, and 30 lx, respectively. The experiments at noon were unsuccessful due to the strong sunlight. Although more powerful projectors [47] could be acquired, we argue that a typical projector is effective in dimmer environments (e.g., cloudy days, at dawn, dusk, and night, or urban areas where buildings cause shades), which accounts for more than half of a day. See Sec. 7.1 for more discussion on ambient lighting conditions.

Results (Tab. 2) of the other cases show that the success rates are 30% lower than our in-lab experiments (the four-meter case from Fig. 11), because we used our in-lab channel model directly in the road experiments without retraining it, and also the environmental conditions are more unpredictable. Moreover, the attack rates on altering some classes (e.g., the STOP sign) into three other signs (e.g., YIELD) were 100%, which is critical as an attacker can easily prevent an autonomous vehicle from stopping at a STOP sign.

6.2.3 Different cameras

Previously, we conducted GhostImage attacks on Aptina MT9M034 camera [34] designed for autonomous driving. Here, we evaluate two other cameras, an Aptina MT9V034 [48] with a simpler lens design, and a Ring indoor security camera [49] for surveillance applications.

Table 2: Outdoor alteration attack success rates

Success rates of	Noon	Dusk	Night
Overall	0%	51%	42.9%
STOP → YIELD	0%	100%	100%
STOP → ADDEDLANE	0%	100%	100%
STOP → PEDESTRIAN	0%	100%	100%

Aptina MT9V034 We mounted system-aware creation attacks against the same camera-based object classification system as in Section 6.1 but we replaced the camera with the Aptina MT9V034 camera. Since this camera has a smaller aperture size and also a simpler lens design than Aptina MT9M034, for a distance of one meter, only 16×16 -resolution attack patterns could be achieved (previously we had 32×32 at one meter). We did not train a new channel model for this camera, so the attack success rate at one meter was only 75%, which is 25% lower than the Aptina MT9M034 camera. As the distances increased up to four meters, creation attacks yielded success rates as 46.25%, 33.75%, and 12.5%, respectively. Another reason why the overall success rate was lower is that even though the data sheet of Aptina MT9V034 [48] states that the camera also has the auto exposure control feature, we could not enable the feature in our experiments. In other words, system-aware creation attacks did not benefit from the exposure control. This, on the other hand, indicates the robustness of GhostImage attacks: Even without taking advantage of exposure control, the attacks were still effective, with attack success rates as high as 75%.

Ring indoor security camera We tested GhostImage untargeted attacks against a Ring indoor security camera [49] on the ImageNet dataset. To demonstrate that our attacks can be applied to surveillance scenarios, we assume the camera would issue an intrusion warning if a specific object type [50] is detected by the Inception V3 neural network [46]. The attacker’s goal is to change an object for an intruder class to a non-intruder class. However, we could not find “human”, “person” or “people”, etc. in the output classes, we instead used five human related items (such as sunglasses) as the

Table 3: GhostImage untargeted alteration attacks against Ring camera on ImageNet dataset in perception domain

Index	Benign Class	Rate	Common Prediction
19992	fur boat	100%	geyser, parachute
21539	sunglasses	100%	screen, microwave
22285	sunglasses	100%	plastic bag, geyser
31664	sarong	100%	jellyfish, plastic bag
2849	sweatshirt	100%	laptop, candle
26236	puncho	100%	table lamp

benign classes. We found six images from the validation set of ImageNet, of which top-1 classification results are one of those five benign classes. The six images were displayed on a monitor. For each benign image, we calculated ten alternative 3×3 attack patterns (the highest resolution at one meter by the Ring camera). Results show that for all six benign images, system-aware attacks achieved untargeted attack success rates of 100% (Table 3).

7 Discussion

In this section, we discuss practical challenges to GhostImage attacks, speculate as to effective countermeasures.

7.1 Practicality of GhostImage Attacks

Moving targets and alignment: The overlap of ghosts and objects of interest in images must be nearly complete for the attacks to succeed. In the cases of a moving camera (e.g., one mounted to a vehicle), the attacker needs to be able to accurately track the movement of the targeted camera, otherwise the attacker can only sporadically inject ghosts. Note that, although aiming (or tracking) moving targets is generally challenging in remote sensor attacks (e.g., the AdvLiDAR attack [12] assumes the attacker can achieve this via camera-based object detection and tracking), existing works [18, 29] have demonstrated the feasibility of tracking cameras and then neutralizing them. This paper’s main goal is to propose a new category of camera attacks, which enables an attacker to inject arbitrary patterns.

Conspicuousness: The light bursts around the light source in Figures 1 and 6 may raise stealthiness concerns about our attacks. However, according to our analysis in Sec. 4.2, such bursts can actually be eliminated because the light source can be outside of view [31]. Even the light source has to be in the frame (due to the lens configuration), we argue that a camera-based object classification system used in autonomous systems generally make decisions without human input (for example, in a Waymo self-driving taxi [1], no human driver is required). Additionally, the attack beam is so concentrated that only the victim camera can observe it while other human-beings (e.g., pedestrians) cannot (Fig. 14). Finally, the light source only needs to be on for a short amount of time, as a few tampered frames can cause incorrect actions [51].

Projectors, lenses, and attack distances: Based on our model (Eq. 7) and experiments (Tab. 4), the illuminance on the camera from the projector would better be $4/3$ of the part from ambient illuminance (to achieve a success rate of 100%). Since $\text{Illuminance} \propto \text{Luminance} \cdot r_{\text{throw}}^2 / d^2$, in order to carry out an attack during sunny days (typically with Illuminance 40×10^3 lx), a typical projector (e.g., [52] with Luminance 9×10^3 lm) should work with a telephoto lens [53] (with a throwing radio 100) at a distance of one meter. For longer distances or brighter backgrounds, one can either acquire a more

powerful projector (e.g., [47] with 75×10^3 lm), or combine multiple lenses to achieve much larger throwing ratios (e.g., two Optela lenses [53] yield 200, etc.), or both.

Knowledge of the targeted system: We assume that both types of attackers know about the camera matrix M_c and color calibration matrix H_c . We note that the attacks can still be *effective* without such knowledge but with it the attacks can be more *efficient*. For example, the attacker may choose to lower their attack success expectation but the probability of successful attack may still be too high for potential victims to bear (e.g., a success rate of only 10% might be unacceptable for reasons of safety in automated vehicles). This challenge can be largely eliminated if the attacker is able to purchase a camera of the same, or similar, model as used in the targeted system and use it to derive the matrices. Although the duplicate camera may not be exactly the same to the target one, the channel model would still be in the same form with approximate, probably fine-tuned parameters (via retraining), thanks to the generality of our channel model. Lastly, assuming white-box knowledge on sensors is widely adopted and accepted in the literature, e.g., the AdvLiDAR attack [12]. Also, we assume white-box attacks on the neural network, though this assumption can be eliminated by leveraging the transferability of adversarial examples [54, 55].

Object detection: We have assumed that the object detector can crop out the region of the image which contains the projected ghost pattern(s). Though it cannot be guaranteed that an object detector will automatically include the ghost patterns, we note that a GhostImage attacker could design ghost patterns that cause an object detector to include them [27, 56] and, at the same time, the cropped image would fool the subsequent object classifier.

Attack Variations: Instead of flare effects, we can also leverage beamsplitting to merge the benign image and the adversarial one together. Rather than projectors, lasers can also be used. Please see our full version [57] for more details.

7.2 Countermeasures

The most straightforward countermeasure to GhostImage attacks is flare elimination, either by using a lens hood or through flare detection. Lens hoods are generally not favored as they reduce the angle of view of the camera, which is unacceptable for many autonomous vehicle and surveillance applications. Adversarial flare detection is challenging as they are typically transparent [58], and hard to be distinguished from natural ghosts.

A complementary line of defense would be to make neural networks themselves robust to GhostImage attacks. Existing approaches against adversarial examples (e.g., [45, 59–61], etc.) are ill-suited for this task, however, as GhostImage attacks do not necessarily follow the constraints placed on traditional adversarial examples in that perturbations do not have to be bounded within a small norm, meanwhile these defenses

were not designed for arbitrarily large perturbations. As this work mainly focuses on sensor attacks, we leave the validation of defenses as future work [12, 25–27].

Another complementary approach of defense is to exploit prior knowledge, such as GPS locations of signs, to make decisions, instead of only depending on real-time sensor perception (though this approach would not work for spontaneous appearance of objects, e.g., in the context of collision avoidance). Sensor redundancy/fusion could also be helpful: autonomous vehicles could be equipped with multiple cameras and/or other types of sensors, such as LiDARs and radars, which would at least increase the cost of the attack by requiring the attacker to target multiple sensors. However, a powerful attacker may be able to attack LiDARs [12], radars [19] and cameras simultaneously to defeat sensor fusion. Finally, temporal consistency via object tracking (e.g., “the object should not have appeared from nowhere of a sudden.”) may also be used to detect the attack, or at least complicate it.

8 Related Work

Sensor attacks Perception in autonomous and surveillance systems occurs through sensors, which convert analog signals into digital ones that are further analyzed by computing systems. Recent work has demonstrated that the sensing mechanism itself is vulnerable to attack and that such attacks may be used to bypass digital protections [15, 16]. For example, anti-lock braking system (ABS) sensors have been manipulated via magnetic fields by Shoukry et al. [62], microphones have been subject to inaudible voice and light-based attacks [9, 63], and light sensors can be influenced via electromagnetic interference to report lighter or darker conditions [8]. The reader is referred to [15, 16] for a review of analog sensor attacks.

Existing remote attacks against cameras [11, 18, 19] are denial-of-service attacks and do not seek to compromise the object classifier as our GhostImage attacks do. Those attacks that do target object classification [25, 27, 64] are either digital or physical domain attacks (i.e., they need to modify the object of interest in this case a traffic sign or road pavement, physically or after the object has been captured by a camera) rather than perception domain attacks [15, 16]. Li et al. [28]’s attacks on cameras require attackers to place stickers on lenses, to which is generally hard to get access. Similarly, several light-based attacks [51, 65, 66] fall within the domain of physical attacks, as opposed to our perception domain attack, because these approaches illuminate the object of interest with visible or infrared light. We did not consider infrared noise in our attacks as it can be easily eliminated from visible light systems using infrared filters. Attacks on LiDAR systems [10–12, 67] are also related; but they are considerably easier to carry out than our visible light-based attacks against cameras because attackers can directly inject adversarial laser pulses into LiDARs without worrying about blocking the object of interest.

Adversarial examples State-of-the-art adversarial examples can be categorized as digital (e.g., [23, 24]), or physical domain attacks (e.g., [25, 26, 68]) in which objects of interest are physically modified to cause misclassification. The latter differs from GhostImage attacks in that we target the sensor (camera) without needing to physically modify any real-world object. Another line of work focuses on unrestricted adversarial examples (so as ours), such as [69], though they are limited in the digital domain.

9 Conclusion

In this work we presented GhostImage attacks against camera-based object classifiers. Using common optical effects, viz. lens flare/ghost effects, an attacker is able to inject arbitrary adversarial patterns into camera images using a projector. To increase the efficacy of the attack, we proposed a projector-camera channel model that predicts the location of ghosts, the resolution of the patterns in ghosts, given the projector-camera arrangement, and accounts for exposure control and color calibration. GhostImage attacks also leverage adversarial examples generation techniques to find optimal attack patterns. We evaluated GhostImage attacks using three image datasets and in both indoor and outdoor environments on three cameras. Experimental results show that GhostImage attacks were able to achieve attack success rates as high as 100%, and also have potential impact on autonomous systems, such as self-driving cars and surveillance systems.

Acknowledgments

The work was partly supported by NSF grants CNS-1801402, CNS-1410000, and CNS-1801611. We would like to thank the anonymous reviewers, and Xiaolan Gu, Mingshun Sun for their helps on this work.

A Illustrative Channel Model Parameters

Table 4 lists all parameters of the projector-camera channel model. The color calibration matrix is

$$H_c = \begin{bmatrix} 0.5 & 0 & 0.1 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.8 \end{bmatrix}.$$

References

- [1] Waymo. Waymo. <https://waymo.com>, 2020.
- [2] Tesla. Autopilot. <https://www.tesla.com/autopilot>, 2020.
- [3] Amazon. Prime air delivery.
- [4] Google. Nest and google home. now under one roof. nest.com, 2020.
- [5] Amazon. Ring. ring.com, 2020.

Table 4: Channel model parameter examples

Description	Symbol	Value
Throwing ratio	r_{throw}	20
Physical size of ghosts	S_f	0.0156 cm ²
Projection resolution	P_O	1024 × 768
Flare booster	ρ	30
Bulb intensity	T_a	[0, 1]
Ambient illuminance	I_{env}	300 lx (indoor)
Projector ill.	I	400 lx (at 1 m)
Projector max ill.	I_{max}	1200 lx (at 1 m)
Camera matrix	M	See below
Color calibration matrix	H_c	See below
In Equation 6	a	8.9
In Equation 6	b	6.7
In Equation 6	c_t	-7.8
In Equation 6	c_d	0.25

- [6] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.
- [7] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.
- [8] Jayaprakash Selvaraj, Gökçen Y Dayanıklı, Neelam Prabhu Gaunkar, David Ware, Ryan M Gerdes, Mani Mina, et al. Electromagnetic induction attacks against embedded systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 499–510. ACM, 2018.
- [9] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection attacks on voice-controllable systems.
- [10] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 445–467. Springer, 2017.
- [11] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 11:2015, 2015.
- [12] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2267–2281. ACM, 2019.
- [13] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 881–896, 2015.
- [14] Qiben Yan, Kehai Liu, Qin Zhou, Hanqing Guo, and Ning Zhang. Surfingattack: Interactive hidden attack on voice assistants using ultrasonic guided wave. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.
- [15] C. Yan, H. Shin, C. Bolton, W. Xu, Y. Kim, and K. Fu. Sok: A minimalist approach to formalizing analog sensor security. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 480–495, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.
- [16] Ilias Giechaskiel and Kasper Bonne Rasmussen. Taxonomy and challenges of out-of-band signal injection attacks and defenses. *IEEE Communication Surveys & Tutorials*, 2020.
- [17] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.
- [18] Khai N Truong, Shwetak N Patel, Jay W Summet, and Gregory D Abowd. Preventing camera recording by designing a capture-resistant environment. In *International conference on ubiquitous computing*, pages 73–86. Springer, 2005.
- [19] Chen Yan, Wenyuan Xu, and Jianhao Liu. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON*, 24, 2016.
- [20] Evan Ribnick, Stefan Atev, Osama Masoud, Nikolaos Papanikolopoulos, and Richard Voyles. Real-time detection of camera tampering. In *2006 IEEE International Conference on Video and Signal Based Surveillance*, pages 10–10. IEEE, 2006.
- [21] Qingquan Li, Long Chen, Ming Li, Shih-Lung Shaw, and Andreas Nüchter. A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios. *IEEE Transactions on Vehicular Technology*, 63(2):540–555, 2013.
- [22] Matthias B. Hullin, Elmar Eisemann, Hans-Peter Seidel, and Sungkil Lee. Physically-based real-time lens flare rendering. *ACM Trans. Graph. (Proc. SIGGRAPH 2011)*, 30(4):108:1–108:9, 2011.
- [23] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.
- [24] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 39–57. IEEE, 2017.
- [25] Kevin Eykholt, Ivan Evtimov, Earlene Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1625–1634, 2018.
- [26] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1528–1540. ACM, 2016.
- [27] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. Seeing isn’t believing: Towards more robust adversarial attack against real world object detectors. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1989–2004. ACM, 2019.
- [28] Juncheng B Li, Frank R Schmidt, and J Zico Kolter. Adversarial camera stickers: A physical camera attack on deep learning classifier. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, 2019.
- [29] KYLE MIZOKAMI. China could blind u.s. satellites with lasers. <https://www.popularmechanics.com/military/weapons/a29307535/china-satellite-laser-blinding/>, 2019.
- [30] Patricia Vitoria and Coloma Ballester. Automatic flare spot artifact detection and removal in photographs. *Journal of Mathematical Imaging and Vision*, 61(4):515–533, 2019.
- [31] Gunawan Kartapranata. Lens flare at borobudur stairs kala arches, 2010.
- [32] Hsien-Che Lee. *Introduction to color imaging science*. Cambridge University Press, 2005.
- [33] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

- [34] ON semiconductor. *MT9M034 1/3-Inch CMOS Digital Image Sensor*.
- [35] NEC. *NP Installation Series User's Manual*, 10 2007.
- [36] Canon. Telephoto zoom ef-s 55-250mm.
- [37] NEC. Np05zl, 4.62–7.02:1 zoom lens.
- [38] Andreas Mogelmoose, Mohan Manubhai Trivedi, and Thomas B Moeslund. Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1484–1497, 2012.
- [39] Yanmao Man, Ming Li, and Ryan Gerdes. Poster: Perceived adversarial examples. In *IEEE Symposium on Security and Privacy*, 2019.
- [40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.
- [41] Nvidia. *NVIDIA TESLA P100 GPU ACCELERATOR*, 2016.
- [42] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387. IEEE, 2016.
- [43] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
- [45] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 582–597. IEEE, 2016.
- [46] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- [47] Barco. Xdl-4k75.
- [48] ON semiconductor. *MT9V034 1/3-Inch Wide-VGA CMOS Digital Image Sensor*, 2017.
- [49] Ring. Indoor security cameras. <https://shop.ring.com/collections/security-cams#indoor>, 2019.
- [50] Ring. Standard and advanced motion detection systems used in ring devices. <https://support.ring.com/hc/en-us/articles/115005914666-Standard-and-Advanced-Motion-Detection-Systems-Used-in-Ring-Devices>, 2020.
- [51] Ben Nassi, Dudi Nassi, Raz Ben-Netanel, Yisroel Mirsky, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Phantom attacks on driver-assistance systems.
- [52] Epson. Pro I1490u wuxga 3lcd laser projector.
- [53] Opteka. Opteka 650-1300mm telephoto zoom lens.
- [54] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.
- [55] Yuxuan Chen, Xuejing Yuan, Jiangshan Zhang, Yue Zhao, Shengzhi Zhang, Kai Chen, and XiaoFeng Wang. Devil's whisper: A general approach for physical adversarial attacks against commercial black-box speech recognition devices. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [56] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlene Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, and Tadayoshi Kohno. Physical adversarial examples for object detectors. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.
- [57] Yanmao Man, Ming Li, and Ryan Gerdes. Ghostimage: Perception domain attacks against vision-based object classification systems. *arXiv preprint arXiv:2001.07792*, 2020.
- [58] Yichao Xu, Hajime Nagahara, Atsushi Shimada, and Rin-ichiro Taniguchi. Transcut: Transparent object segmentation from a light-field image. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3442–3450, 2015.
- [59] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.
- [60] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [61] Tong Wu, Liang Tong, and Yevgeniy Vorobeychik. Defending against physically realizable attacks on image classification. In *8th International Conference on Learning Representations (ICLR)*, 2020.
- [62] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 55–72. Springer, 2013.
- [63] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 103–117, New York, NY, USA, 2017. Association for Computing Machinery.
- [64] Alesia Chernikova, Alina Oprea, Cristina Nita-Rotaru, and BaekGyu Kim. Are self-driving cars secure? evasion attacks against deep neural networks for steering angle prediction. In *IEEE Security and Privacy Workshop on IoT*. IEEE, 2019.
- [65] Zhe Zhou, Di Tang, Xiaofeng Wang, Weili Han, Xiangyu Liu, and Kehuan Zhang. Invisible mask: Practical attacks on face recognition with infrared. *arXiv preprint arXiv:1803.04683*, 2018.
- [66] Luan Nguyen, Sunpreet S. Arora, Yuhang Wu, and Hao Yang. Adversarial light projection attacks on face recognition systems: A feasibility study, 2020.
- [67] James Tu, Mengye Ren, Siva Manivasagam, Ming Liang, Bin Yang, Richard Du, Frank Cheng, and Raquel Urtasun. Physically realizable adversarial examples for lidar object detection, 2020.
- [68] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *Artificial Intelligence Safety and Security*, pages 99–112. Chapman and Hall/CRC, 2018.
- [69] Yang Song, Rui Shu, Nate Kushman, and Stefano Ermon. Constructing unrestricted adversarial examples with generative models. In *Advances in Neural Information Processing Systems*, pages 8312–8323, 2018.

PLC-Sleuth: Detecting and Localizing PLC Intrusions Using Control Invariants

Zeyu Yang
Zhejiang University

Liang He
University of Colorado Denver

Peng Cheng
Zhejiang University

Jiming Chen
Zhejiang University

David K.Y. Yau
Singapore University of Technology and Design

Linkang Du
Zhejiang University

Abstract

Programmable Logic Controllers (PLCs) are the ground of control systems, which are however, vulnerable to a variety of cyber attacks, especially for networked control systems. To mitigate this issue, we design *PLC-Sleuth*, a novel non-invasive intrusion detection/localization system for PLCs, grounding on a set of control invariants — i.e., the correlations between sensor readings and the concomitantly triggered PLC commands — that exist pervasively in all control systems. Specifically, taking the system’s Supervisory Control and Data Acquisition log as input, *PLC-Sleuth* abstracts/identifies the system’s control invariants as a control graph using data-driven structure learning, and then monitors the weights of graph edges to detect anomalies thereof, which is in turn, a sign of intrusion. We have implemented and evaluated *PLC-Sleuth* using both a prototype of Secure Ethanol Distillation System (SEDS) and a realistically simulated Tennessee Eastman (TE) process.

1 Introduction

Background. Commodity *Programmable Logic Controllers* (PLCs) are proved vulnerable to cyber attacks [1, 2]. Researchers have identified >36.7K PLCs that can be accessed by scanning the ports of common communication protocols, such as Modbus and Siemens S7 [3, 4]. Symantec has also confirmed the feasibility of hijacking a number of mission-critical PLCs [5], such as acquiring credentials of the target PLC to administer destructive payloads. Keliris and Maniatakos have designed an autonomous process to compromise PLCs [6], facilitating the executable program injection [7] or firmware modification [8]. After compromising the PLC, adversaries can mount a variety of attacks, including but not limited to:

- *Command Injection Attacks.* The malware TRITON was launched remotely in Saudi Arabia in 2017 [9] to disturb the operations of safety actuators in a petrochemical facility. These actuators were originally designed to take remedial actions in case of emergency, while TRITON instead sent them adversarial commands to shut down the facilities.
- *Cooperative Stealthy Attacks.* To hide attacks from being detected, adversaries can even mount advanced stealthy attacks to PLCs, by not only tampering control commands,

but also cooperatively forging the Supervisory Control and Data Acquisition (SCADA) logs with historical (and normal) data. An example of this stealthy attack is the worm Stuxnet which damaged hundreds of centrifuges [10]. A firmware vulnerability of Allen Bradley PLCs exposed in 2017 [11] allows modifying the PLCs’ control commands and forging sensor readings.

Protecting PLCs Using Control Invariants. To mitigate the above issues, we design a non-invasive data-driven intrusion detection system (IDS) for PLCs, which we call *PLC-Sleuth*. The foundation of *PLC-Sleuth* is a set of *control invariants* that exist pervasively in all control systems: the control commands issued by PLCs correlate strongly with the concomitant sensor readings. *PLC-Sleuth* automatically identifies these control invariants using system’s SCADA logs, and abstracts them as a control graph, in which the nodes represent system variables, such as commands, sensor readings, and control setpoints, and the weights of edges quantify the strength of correlations between system variables. *PLC-Sleuth* then captures the normal behavior of the weights of graph edges using data-driven approaches, and detects, at runtime, the anomalies — edges with abnormal weights indicate the control channels between corresponding system variables have been compromised. *PLC-Sleuth* has the following salient properties.

- *A Cyber-Physical IDS.* *PLC-Sleuth* is built on a set of physically-induced invariants of control systems, i.e., a given actuation is always triggered by specific real-time sensor measurements, which can be observed as the correlations between the time series of issued control commands and the concomitant sensor readings. This physically-induced correlation (i.e., control invariant) makes *PLC-Sleuth* *reliable* as the correlation will not change unless the control rules are updated, and *pervasively deployable* because such control invariants exist in all control systems.
- *A Non-Invasive IDS.* Taking the SCADA logs of a control system as input, *PLC-Sleuth* automatically identifies the control invariants, and uses them to protect the PLCs thereof, via data-driven approaches without probing/perturbing the system, i.e., *PLC-Sleuth* is non-invasive and thus easy to deploy.
- *An IDS Localizing Intrusions.* *PLC-Sleuth*, besides detecting intrusions at PLCs, also localizes the compromised con-

control loops thereof, facilitating swift repair/forensics of the PLCs; the control system otherwise remains unreliable no matter how well the intrusions are detected.

We have implemented and evaluated PLC-Sleuth using our Secure Ethanol Distillation System (SEDS) prototype and a representative/realistic TE chemical process. We first evaluate PLC-Sleuth’s identification of control invariants — in the form of a control graph — using various volumes of training data. We then evaluate PLC-Sleuth’s intrusion detection/localization using the constructed control graph against command injection attacks and cooperative stealthy attacks, with various deviations from the normal commands. The results show that for SEDS and TE respectively, PLC-Sleuth identifies system invariants with {100%, 98.11%} accuracy, detects PLC attacks — even those change the normal commands by only 0.12% — with {98.33%/0.85%, 100%/0%} true/false positives, and localizes compromised control loops with {93.22%, 96.76%} accuracy.

2 Preliminaries and Basic Idea

Here we present the necessary background of control systems and the basic idea of PLC-Sleuth, using our prototype of an ethanol distillation control system, called SEDS (Secure Ethanol Distillation System), as an example. Distillation is the process of separating constituents in a liquid mixture based on the differences in their volatility. SEDS is a scaled-down but fully operational distillation plant, which purifies alcohol from water and is capable of producing alcohol with a purity of 90%.

2.1 SEDS Prototype

A typical control system has four main components organized as feedback control loops [12]:

- A *physical process*, e.g., an ethanol distillation process;
- A set of *sensors* that measure the physical states y of the process, e.g., the temperature and liquid level of the relevant distillation tower;
- A set of *controllers* that generate a set of control commands u based on the control error $x := s - y$, i.e., the difference between sensor reading y and the expected setpoint s ;
- A set of *actuators* that operate as instructed by the controllers’ commands u , e.g., the valves controlling the feeding of materials.

Fig. 1(a) shows our prototype of SEDS, implemented using a control network consisting of sensors, actuators and a SIMATIC S7-300 PLC, as shown in Fig. 1(b). The PLC is equipped with a 315-2 PN/DP CPU, one 32bits DI module, two 8×13 bits AI modules, one 32 bits DO module, and one 8×12 bits AO module. SEDS’s operation is monitored/logged by a WinCC SCADA system, including 3 set-points, 11 sensor readings, and 3 control commands. The

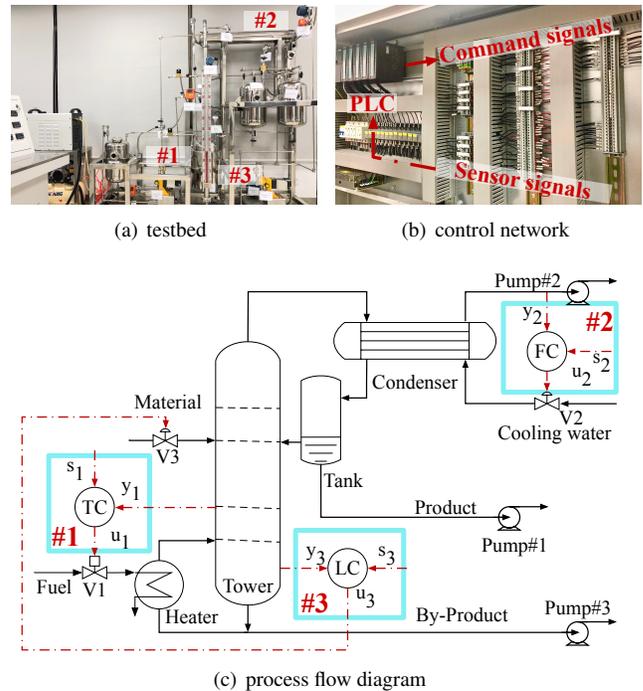


Figure 1: The Secure Ethanol Distillation System (SEDS) prototype in our lab.

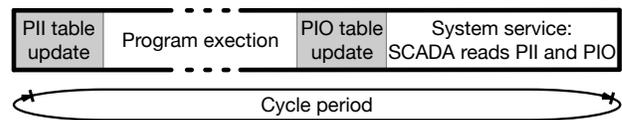


Figure 2: Sequence of CPU executions in PLC [13].

control rules of SEDS are programmed with ladder logic using the Siemens Step7 software. SEDS purifies alcohol from the water with two basic operations — i.e., tower boiling and condenser cooling — realized using three feedback loops, as shown in Fig. 1(c).

- **Loop-1** regulates the tower temperature to a pre-defined level s_1 , at which SEDS achieves a high separation efficiency. The temperature controller (TC) generates a command u_1 based on the difference between s_1 and temperature y_1 collected using a temperature sensor (i.e., $x_1 = s_1 - y_1$), to control the opening/closing of valve V_1 (and hence the fuel flow) to regulate the tower temperature to s_1 .
- **Loop-2** maintains the flow of the condenser’s cooling water — monitored with a liquid level sensor y_2 — around a pre-defined level s_2 to condense ethanol from gas to liquid, which is then refluxed to the tower to improve the distillation concentration further. Both excessive or deficient ethanol refluxing impede the heat/mass transfer and degrade the distillation quality. Also, an unstable flow of cooling water may damage the pump physically. The flow

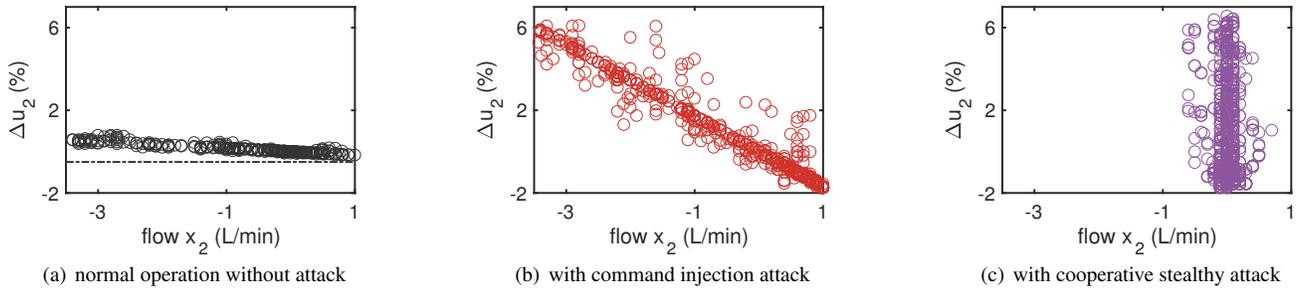


Figure 3: Correlations between SEDS’s command u_2 and flow difference x_2 .

controller (FC) sends a command u_2 to the valve V_2 to regulate the cooling water based on $x_2 = s_2 - y_2$.

- **Loop-3** controls the level of the tower’s liquid material (i.e., sensor reading y_3) to an optimized setpoint s_3 , which is also affected by the ethanol reflux in Loop-2. An unstable liquid level induces premature flood and thus degrades distillation efficiency [14]. Similar to the TC and FC, the level controller (LC) of Loop-3 generates a command u_3 based on the difference $x_3 = s_3 - y_3$ and sends it to valve V_3 to control the distillation tower’s liquid level, by regulating the input flow.

The specific rules of these control loops are implemented in SEDS’s PLC, using the Bang-Bang and Proportional-Integral-Derivative (PID) algorithms, which determine the control signals u based on the current and historical control errors x [15, 16]. In short, SEDS controls system states $\{y_1, y_2, y_3\}$ based on setpoints $\{s_1, s_2, s_3\}$ using control commands $\{u_1, u_2, u_3\}$. These interactions among sensors, setpoints and control commands form the basis of all control systems. At the same time, sensors $\{y_4, \dots, y_{11}\}$ provide more information for system monitoring.

2.2 Attack Model

We consider the following attack model targeting at the PLC of a given control system:

- The attacker can mount the command injection attack during the program execution period (see Fig. 2) by downloading malicious code to PLC.
- Atop the injection attack, the attacker can deliver cooperative stealthy attacks by further replaying normal sensor readings to the PLC’s process-image input (PII) table. Note that SCADA reads sensor readings from PII table, which happens after the execution of malicious code (see Fig. 2). This way, the attacker can deceive the system monitor to conclude a normal operation even if a successful attack has been launched [10].
- The attacker cannot modify the record of actual commands issued during system operation, i.e., any forged commands issued by the attacker will be logged as they are. This is because PIO table logs all commands and remains unchanged

after the execution of the program [17]. By reading from the PIO table (see Fig. 2), SCADA obtains control commands that are actually issued.

We next use a cooperative stealthy attack mounted at SEDS to motivate PLC-Sleuth. The forged commands disturb SEDS’s normal operation, while the faked system logs deceive the system monitor to conclude a normal system operation during/after the attack. Specifically, let us consider the case that an attacker aims to degrade the distillation quality by hacking SEDS’s Loop-2 to trigger an unstable reflux. The increment of command u_2 under a normal operation is proportional to x_2 , i.e., a $x_2 := (s_2 - y_2)$ L/min flow difference triggers the adjustment of the valve by

$$\Delta u_2 = -0.01(\Delta x_2 + 0.17 \times x_2). \quad (1)$$

By exploiting the PLC’s vulnerabilities, e.g., modifying the `s7otbxdx.dll` file of Step7 [10], the attacker downloads a payload to the PLC and implement a malicious control rule of

$$\Delta u_2^a = -0.1(\Delta x_2 + 0.17 \times x_2), \quad (2)$$

thus degrading the stability of the cooling process and causing oscillations in ethanol reflux. Note that the tower level y_3 will also be affected by this hacked Δu_2^a due to an unstable ethanol reflux, degrading the distillation quality further. Also, to hide the abnormal deviations of $\{y_2, y_3\}$, the attacker replaces the sensor logs of $\{y_2^a, y_3^a\}$ with historical steady-state records, by overwriting the PLC’s process-image table. This way, the sensor logs (i.e., $\{y_2, y_3\}$) appear normal to the operator whereas the SEDS is actually manipulated by the attacker.

2.3 Basic Idea of PLC-Sleuth

As stated above, the control command u is generated based on the control error $x = s - y$, using the pre-defined control rules written in the PLC. This incurs strong/reliable correlations between u and x , which we use as the system’s *control invariants*. Fig. 3 visualizes such a control invariant between Δu_2 and x_2 : when SEDS is attacked as described above, the control invariant in Fig. 3(a) changes noticeably due to the malicious control rule in Eq. (2), for both the command injection attack (Fig. 3(b)) and the cooperative stealthy attack

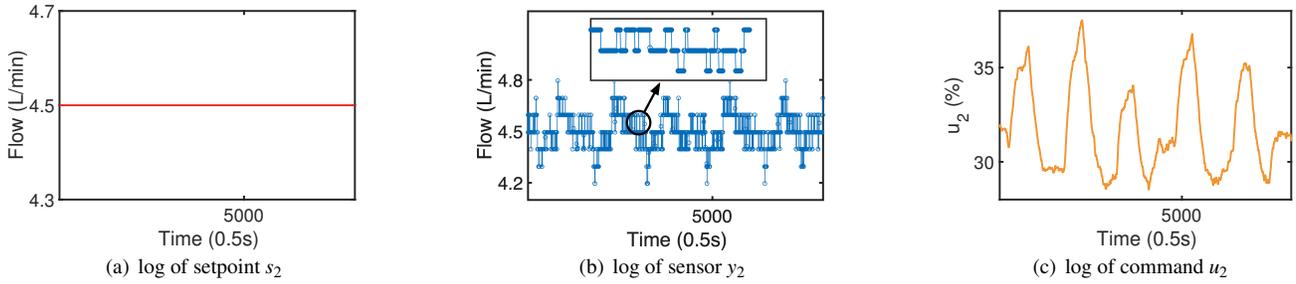


Figure 4: Different patterns of setpoint, sensor, command variables in SEDS’s Loop-2.

(Fig. 3(c)). PLC-Sleuth uses these control invariants to detect PLC intrusions, by detecting deviations of these control invariants from their normal behaviors. The challenge is, in turn, to identify/abstract the system’s control invariants, which PLC-Sleuth addresses using a control graph.

3 Abstracting Control Invariants

For a given control system, PLC-Sleuth identifies and abstracts its control invariants — defined by the system variables and the interactions thereof — using a *control graph*. Specifically, the control graph is defined as a weighted and directed acyclic graph $\mathcal{G}(V, E, W)$, where: (i) V is the set of nodes representing system variables (i.e., setpoints, sensor readings, and commands), (ii) E is the set of directed edges connecting nodes in V , representing the correlations among system variables, and (iii) W is the set of edge weights, representing the strength of the correlations described by E . Specifically,

- $V = \{S, Y, U\}$ consists of a setpoint node set S , a sensor node set Y , and a command node set U , which can be obtained/identified automatically from the SCADA logs. Note that in control systems, the number of nodes in U is no less than that in S , i.e., $|U| \geq |S|$.
- $E = \{E_s^y, E_y^u\}$ consists of a control error edge set E_s^y connecting nodes in S to nodes in Y , and a control command edge set E_y^u connecting nodes in Y to nodes in U . An edge exists in E when its two nodes belong to the same control loop. The commonly used decentralized control scheme — one sensor is fed back for the generation of one targeted control command, and vice versa [18] — makes both the out-degree of nodes in Y and the in-degree of nodes in U equal to 1.
- $W = \{W_s^y, W_y^u\}$ consists of an error weight set W_s^y for control error edges in E_s^y and a command weight set W_y^u for control command edges in E_y^u . PLC-Sleuth exploits W_y^u to detect and localize PLC intrusions, by monitoring W_y^u in real time and detecting the anomalies thereof. Note that the error weight set W_s^y could also be used to detect PLC attacks/anomalies, i.e., the sensor readings Y should be close to the corresponding setpoint S in a stable control

system [12]. This approach, however, is (i) vulnerable to replayed sensor logs in cooperative stealthy attacks [19], and (ii) orthogonal to PLC-Sleuth and thus not discussed here. Also, the fact that \mathcal{G} includes two types of edges/weights differs it from the traditional structure learning graphs [20, 21].

Next we explain PLC-Sleuth’s classification of nodes and edges. For the ease of description, we use s_i , y_j , and u_k to denote nodes belonging to S , Y , and U , respectively, and use $e_{s_i}^{y_j}$ and $e_{y_j}^{u_k}$ to represent edges belonging to E_s^y and E_y^u .

Setpoint Node Set S . A setpoint node $s_i \in S$ denotes an expected state of system, which will not fluctuate when the system is operating, as shown in Fig. 4(a) and Figs. 20(a)-20(c) in Appendix.

Sensor Node Set Y . A sensor node $y_j \in Y$ captures the system’s real-time state. Because of the environment noise and the sensor measurement error, the readings of y_j show small but consistent vibrations, as observed in Fig. 4(b) and Figs. 20(d)-20(n) in Appendix.

Command Node Set U . A command node $u_k \in U$ reflects the actuation to achieve the pre-defined system state, and fluctuates with the readings of the corresponding sensor. The issued command u_k will be smooth to protect the actuator, as shown in Fig. 4(c) and Figs. 20(o)-20(q) in Appendix.

Error Weight Set W_s^y . The weight of control error edge $w_{s_i}^{y_j} \in W_s^y$ captures the difference between system state y_j and the corresponding setpoint s_i over the recent l samples,

$$w_{s_i}^{(y_1, \dots, y_d)} = \left| \sum_{t=1}^l (s_i(t) - f(y_1(t), \dots, y_d(t))) \right|, \quad (3)$$

where d is the number of sensors used to estimate the system state using function f , which can be obtained from control algorithms’ expressions or binary files [22].

Command Weight Set W_y^u . Mutual information (MI) — a metric commonly used to characterize conditional dependency between variables using their posterior probability distribution [23] — is an intuitive metric to define command weight $w_{y_j}^{u_k} \in W_y^u$. Specifically, the weight of command edge can be defined as the normalized mutual information, i.e.,

$$w_{y_j}^{u_k} = \frac{I(x_i, u_k)}{H(u_k)}, \quad (4)$$

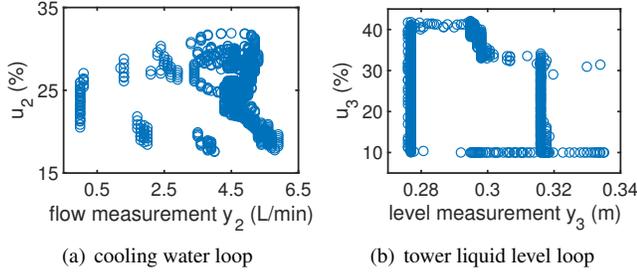


Figure 5: Static correlations (calculated according to Eq. (4)) between SEDS's commands and sensor readings.

where x_i is the control error between s_i and y_j , $H(u_k)$ is the entropy of the recent sequence of commands u_k , and $I(x_i, u_k)$ denotes the MI between x_i and u_k , which is calculated using the joint probability density function of variable x_i and u_k (denoted as $p(\xi, \eta)$) as

$$I(x_i, u_k) = \sum_{\xi \in X} \sum_{\eta \in U} p(\xi, \eta) \log \left(\frac{p(\xi, \eta)}{p(\xi)p(\eta)} \right). \quad (5)$$

However, because a variable's posterior probability only depicts static information, MI has limited ability to quantify the dynamic correlation between sensor measurement y and command u . Fig. 5 shows the scatter plot of SEDS's sensor readings $\{y_2, y_3\}$ and control commands $\{u_2, u_3\}$, showing their low correlation when only considering samples collected at a given time instant — the weighting scheme in Eq. (4) may not work well for PLC-Sleuth.

To mitigate the above limitation, we define a novel transition correlation, using the correlation between the two time series of Δu and y , as visualized in Fig. 6 where the close-to-diagonal path indicates much stronger correlation (when compared to Fig. 5).

Specifically, inspired by the fact that the command u_k is triggered according to the current and historical values of its corresponding error series $x_i = s_i - y_i$, we define the *transition mutual information (TMI)* as,

$$TMI(x_i, \Delta u_k) = \sum_{\xi \in X^\tau} \sum_{\eta \in U^1} p(\xi, \eta) \log \left(\frac{p(\xi, \eta)}{p(\xi)p(\eta)} \right), \quad (6)$$

where τ is the length of sequences used for characterizing variables' transitions. This way, PLC-Sleuth calculates the command weight $w_{y_j}^{u_k}$ as

$$w_{y_j}^{u_k} = \frac{TMI(x_i, \Delta u_k)}{H(\Delta u_k)}. \quad (7)$$

The weighting of MI and TMI are compared statistically in Table 1 of Sec. 5.

Control Graph Example. As an example, Fig. 7 shows the control graph $\mathcal{G}(V_{SEDS}, E_{SEDS}, W_{SEDS})$ of SEDS, where

$$\begin{aligned} V_{SEDS} &= \{\{s_1, s_2, s_3\}, \{y_1, y_2, y_3\}, \{u_1, u_2, u_3\}\}, \\ E_{SEDS} &= \{\{e_{s_1}^{y_1}, e_{s_2}^{y_2}, e_{s_3}^{y_3}\}, \{e_{y_1}^{u_1}, e_{y_2}^{u_2}, e_{y_3}^{u_3}\}\}, \\ W_{SEDS} &= \{\{w_{s_1}^{y_1}, w_{s_2}^{y_2}, w_{s_3}^{y_3}\}, \{w_{y_1}^{u_1}, w_{y_2}^{u_2}, w_{y_3}^{u_3}\}\}. \end{aligned}$$

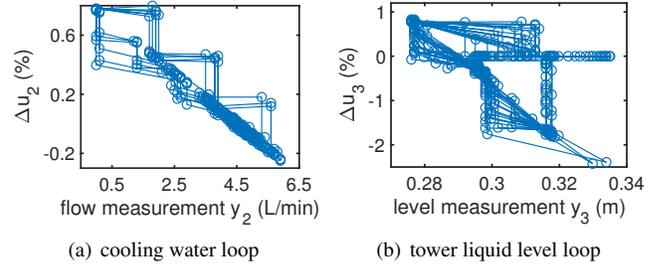


Figure 6: Transition correlations (calculated according to Eq. (7)) between SEDS's commands and sensor readings.

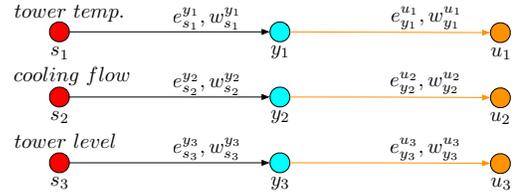


Figure 7: Control graph of SEDS.

Note that, again, both the out-degree of $\{y_1, y_2, y_3\}$ and the in-degree of $\{u_1, u_2, u_3\}$ are 1. The sensor variables $\{y_4, \dots, y_{11}\}$, which are used for system monitoring only, are not shown in Fig. 7 for clarity.

4 Design of PLC-Sleuth

Fig. 8 presents the logic flow of PLC-Sleuth: constructing the control graph $\mathcal{G}(V, E, W)$ of the system-of-interest by identifying the variable's connections (i.e., edges E) and the corresponding weight W , and monitoring the time series of weight W_y^u to detect and localize PLC attacks.

4.1 Construction of Control Graph

An intuitive way to construct the control graph is to have system designers manually extract it from system documents (e.g., engineering flow diagrams, loop diagrams, logic diagrams, electrical control diagrams, etc.). However, this requires significant human efforts or field-expertise and is error-prone [24]. As an alternative, PLC-Sleuth constructs the control graph \mathcal{G} automatically with a data-driven approach, using the historical (and normal) SCADA logs.

4.1.1 Identifying Vertex Set

PLC-Sleuth identifies the nodes of \mathcal{G} automatically using the SCADA logs: (i) setpoint variables are of constant values, (ii) sensor readings are of consistent and small vibrations, and (iii) control commands are of continuous/smooth values, as shown in Fig. 4. PLC-Sleuth first identifies the setpoint node set S by finding the constant variables (i.e. $H(s_i) = 0$). Then by calculating the *vibrations-signal ratio (VSR)* of variable y_j , PLC-Sleuth identifies sensor node set Y from the variables

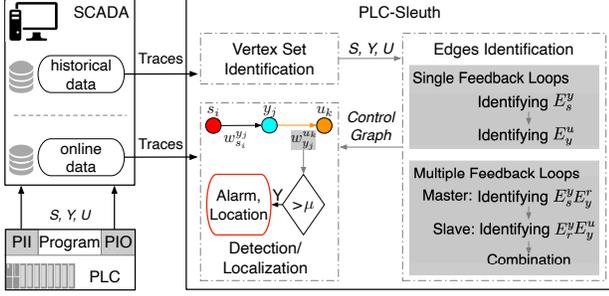


Figure 8: Architecture of PLC-Sleuth.

with consistent fluctuation. Here we define vibrations as the curve crests/troughs $y_j(k)$ in y_j , where $y_j'(k) \times y_j'(k+1) < 0$, and VSR_{y_j} as the maximum ratio of crests/troughs in a period samples of y_j . With a 10 persistent samples setting, all sensor nodes in SEDS have the VSR of 1, as shown in Fig. 9. After selecting nodes with high VSR (larger than 0.5 in Fig. 9) as sensor node set, the unclassified variables are classified as the command node set U .

4.1.2 Identifying Edge Set for PLCs with Single Feedback Loops

On top of the identified nodes, PLC-Sleuth has to identify the edges and determine their weights. For the ease of description, let us first consider the construction of control graphs for PLCs consisting of loops with only a single feedback channel, e.g., the three control loops in SEDS. We will later extend the graph construction to more complex control loops involving multiple (and likely coupled) feedback channels.

• **Step-I: Identifying E_s^y Edges.** PLC-Sleuth first identifies the error edges in E_s^y , which allows the identification of command edges in E_y^u later. It is desirable for control systems to operate at a steady state that leads to a high system efficiency, which is achieved in practice by using a set of setpoints for each feedback loop of the PLC. Hence, we expect the sensor reading y to be close to its corresponding setpoint s , which steers PLC-Sleuth's identification of edges in E_s^y . The basic idea is that, for each node $\tilde{s}_i \in S$, we identify its corresponding edge in E_s^y by finding the node $\tilde{y}_j \in Y$, such that the weight $w_{\tilde{s}_i}^{\tilde{y}_j}$ (calculated with Eq. (3)) is the smallest among all the $\{\tilde{s}_i, y_j\}$ pairs. Taking SEDS as an example, $\{s_1, s_2, s_3\}$ are set as $\{51^\circ\text{C}, 4.5\text{L/min}, 0.29\text{m}\}$, and the sensor readings $\{y_1, y_2, y_3\}$ fluctuate around $\{51, 4.5, 0.29\}$. As a result, the weights of $\{w_{s_1}^{y_1}, w_{s_2}^{y_2}, w_{s_3}^{y_3}\}$ tend to be small positive values.

After identifying the edge connecting y_j to s_i , PLC-Sleuth obtains the corresponding time series of control error $x_i(t) = s_i(t) - y_j(t)$, which are used to identify the command edges in E_y^u , as we explain next.

• **Step-II: Identifying E_y^u Edges.** For each of the above identified error edges $e_{s_i}^{y_j} \in E_s^y$, PLC-Sleuth further matches its sensor error node y_j to the command node u_k of the same control loop. Typically, a control algorithm regulates an actuator's

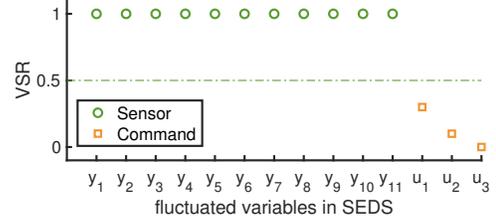


Figure 9: VSR of fluctuated variables in SEDS.

action based on sensor measurements to minimize the control error, making a command node u_k closely correlated with the sensor reading y_j of the same control loop. We use Eq. (7) to quantify the correlation between nodes y_j and u_k , based on the identified E_s^y . Similar to the identification of edges in E_s^y , we match a command node \tilde{u}_k to the sensor node \tilde{y}_j of an identified measurement edge $e_{s_i}^{\tilde{y}_j}$, by finding the maximal $w_{\tilde{y}_j}^{\tilde{u}_k}$ (calculated with Eq. (7)) in all the $\{\tilde{y}_j, u_k\}$ pairs. This way, PLC-Sleuth matches each sensor node \tilde{y}_j of the identified edge in E_s^y with the most correlated command node $\tilde{u}_k \in U$.

• **Step-III: Rematching of Repeated E_y^u Edges.** A control command u operates an actuator in the same control loop. However, the operated actuator does not only affect control loop to which it belongs, but it may also affect other control loops due to their physical interactions. For example, the command u_3 in SEDS does not only affect the tower liquid level y_3 , but also the tower temperature y_1 , because more intake of cold materials will cool the tower more. As a result, both y_1 and y_3 are likely to be matched with u_3 . In general, if a control system has two or more closely coupled components, it may happen that a command node u_k^* is matched to multiple sensor nodes y_j s of different $e_{s_i}^{y_j}$ s edges, thus violating that the in-degree of a command node should be 1. We call the command node u_k^* being matched to multiple y_j s the *repeated command node*. For a repeated command node u_k^* , we delete its edges from \mathcal{G} by keeping only the edge with the maximal weight. An example of the repeated node is u_3 of SEDS, which could be potentially matched to both y_1 and y_3 . PLC-Sleuth eventually removes $e_{y_1}^{u_3}$ because $w_{y_1}^{u_3} < w_{y_3}^{u_3}$.

After addressing all repeated command nodes, PLC-Sleuth repeats Step-II and III for the remaining unmatched y_j and u_k , until all edges in E_s^y , or their corresponding sensor reading nodes y_j s more specifically, are matched with a command node u_k .

4.1.3 Identifying Edge Set for PLCs with Multiple Feedback Loops

Many real-world control systems use control loops with coupled feedback channels — such as cascade control and ratio control [18, 25] — to improve their efficiency. These coupled loops usually operate in a master-slave manner, as illustrated in Fig. 10. Taking SEDS as an example, to stabilize the reflux temperature at a setpoint level, we can deploy another temper-

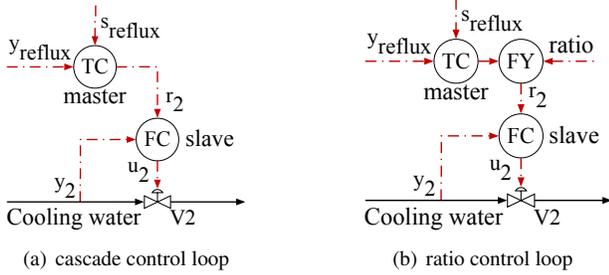


Figure 10: Examples of coupled feedback control schemes.

ature controller TC to SEDS’s Loop-#2 to adjust, in real-time, FC’s setpoint, making FC the slave controller of TC.

Transfer node $r_m \in R$, where $R \subset U$, is introduced to construct the control graph with coupled feedback channels, which is responsible for transferring control commands from master to slave loops (e.g., r_2 in Fig. 10). Note that most transfer nodes also act as setpoints of the slave loops. We refer to transfer nodes that do not act as the setpoints of slave loops as transmit nodes. PLC-Sleuth identify R from U . Specifically, each control loop of a PLC generates a corresponding command, all of which form U . Also, only a subset of U are sent to actuators in the coupled feedback control¹, and the remaining commands form the transfer node set R .

In the following, we explain how PLC-Sleuth constructs the control graph for these multiple channels feedback loops.

• **Step-IV: Constructing the Master Loops.** Master loops are those containing a setpoint node in S . We construct a control graph for the master loops from node set $\{S, Y, R\}$. The identification of error edges in E_s^y of master loops is done in the same way as Step-I. Also, the command edges in E_y^r in master loops are identified similarly as in Step-II and III, by replacing node $u_k \in U$ with node $r_m \in R$.

• **Step-V: Constructing the Slave Loops.** The control graph for slave loops is constructed from node set $\{R, Y$ and $U - R\}$. Error edges in E_r^y of slave loops are identified similarly to Step-I, by replacing Eq.(3) with $w_{r_m}^{(y_1, \dots, y_d)} = \min\{|\sum_{t=1}^L (r_m - f(y_1, \dots, y_d))|, |\sum_{t=1}^L (r_m * r_n - f(y_1, \dots, y_d))|\}$. This step identifies the sensor node \tilde{y}_j (and \tilde{r}_n as well if the ratio control scheme exists) for the transfer node \tilde{r}_m . Transfer nodes that are not matched with a sensor node are treated as transmit nodes. Based on the identified edges from set R to set Y , command edges in E_y^u of slave loops are identified with the same approach as in Step-II and III.

• **Step-VI: Combining the Control Loops.** Master and slave loops are combined using transfer nodes in R . When a sensor node \tilde{y}_j in the master loop is matched to a transmit node \hat{r}_n , which transfers the control command but does not act as a slave setpoint, we further match \hat{r}_n with a slave setpoint node \tilde{r}_m with the maximal $w_{\hat{r}_n}^{\tilde{r}_m}$ (calculated with Eq. (7)) in all the $\{\hat{r}_n, r_m\}$ pairs. For the command edge

¹ Actuators will send received commands back to SCADA system [26,27].

$e_{\tilde{y}_j}^{\tilde{r}_m}$ of master loop not constructed by the transmit node, the sensor node \tilde{y}_j is matched directly to the setpoint node \tilde{r}_m of the slave loop. This process terminates when each of the transfer node r_m of the command edge $e_{y_j}^{r_m}$ in master loop is matched with a slave loop. Then, by comparing the weights of $w_{\tilde{y}_j}^{\tilde{r}_m}$ and $w_{\tilde{y}_j}^{\hat{r}_n}$ (or $w_{\tilde{y}_j}^{\tilde{r}_n}$), we determine the specific control graph (i.e., single feedback loops or multiple feedback loops) to which the nodes $\tilde{s}_i, \tilde{y}_j, \tilde{r}_m, \hat{r}_n, \tilde{u}_k$ belong.

4.2 Detection/Localization of PLC Intrusions

PLC-Sleuth then detects/localizes, at runtime, PLC intrusions using an online norm model constructed using \mathcal{G} .

As illustrated in Sec. 2.3, tampering the control command will violate the control invariants between commands and the corresponding measurements. The violations of control invariants will be observed as the changes of weights in $\mathcal{G}(V, E, W)$. PLC-Sleuth uses a sliding window T to construct an online detector that monitors the weights of edges in E_y^u , by:

$$w_{y_j}^{u_k}(t) = \frac{I(x_i([t-T, t]), \Delta u_k([t-T, t]))}{H(\Delta u_k([t-T, t]))}, \quad (8)$$

where $x_i([t-T, t])$ corresponds to control errors of $y_j(t)$ in window T and $u_k([t-T, t])$ denotes the PLC’s control commands in the same window.

Intrusion Detection. PLC-Sleuth uses a memory-based method, such as the nonparametric CUMulative SUM (CUSUM) [28], to alarm operators if an anomaly is detected in Eq.(8). CUSUM is defined recursively as

$$S_0 = 0 \quad \text{and} \quad S_t = \max(0, S_{t-1} + |v_{t-1}| - \delta), \quad (9)$$

where v_t is the weight error from the expected value $\hat{w}_{y_j}^{u_k}(t)$, defined as

$$v_t = w_{y_j}^{u_k}(t) - \hat{w}_{y_j}^{u_k}(t), \quad (10)$$

and δ is a small positive constant, set as $\delta > |v_{t-1}|$ when the system operates normally, preventing S_t from increasing persistently. An alarm is triggered whenever S_t is larger than a pre-defined threshold, i.e.,

$$S_t > \mu, \quad (11)$$

at which time the detection will be reset with $S_t = 0$.

Intrusion Localization. It is trivial to localize the forged command if only one abnormal edge is detected, i.e., the command responsible for that edge is compromised. When multiple anomalies are detected, PLC-Sleuth localizes the forged command as the one triggering the anomaly alarm first. This greedy strategy is intuitive because cascaded anomalies require a longer time to cause instability to control systems, when compared to the directly forged command. This can be well-justified using SEDS as an example: the forged u_2^c in Eq. (2) first degrades the stability of the ethanol reflux in Loop-#2, causing alarms at edge $e_{y_2}^{u_2}$, and then further oscillates the

Table 1: Weight of SEDS’s command edges obtained with different weighing schemes.

Metric	K2			LL			MDL			MI			PLC-Sleuth			
Score	u_1	u_2	u_3	u_1	u_2	u_3	u_1	u_2	u_3	u_1	u_2	u_3	u_1	u_2	u_3	
	y_1	38918	37101	145	-31778	-27892	-6405	5183	13628	82	0.04	0.02	0.03	0.75	0.08	0.12
	y_2	39006	42114	37	-31253	-20940	-6484	-4256	10942	-100	0.06	0.27	0.02	0.14	0.85	0.40
	y_3	41192	37648	5373	-24187	-24796	-787	-12723	-9470	5420	0.27	0.13	0.88	0.50	0.51	0.97

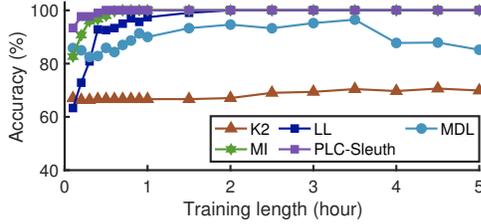


Figure 11: Constructing SEDS’s control graph.

liquid level y_3 in Loop-#3 through fluid transportation, which may induce alarms at edge $e_{y_3}^{u_3}$ after the first alarm is triggered. This way, PLC-Sleuth localizes the forged command as the one controlling Loop-#2.

5 Evaluation

We have evaluated PLC-Sleuth using both SEDS and a simulated Tennessee Eastman (TE) process, which is a representative benchmark of continuous chemical processes [29].

5.1 Methodology

Our evaluation of PLC-Sleuth consists of two parts. We first evaluate PLC-Sleuth’s construction of the control graph by comparing it with the cases when defining $w_{y_j}^{u_k} \in W_y^u$ using four other metrics proposed in structure learning: (i) Bayesian scoring metric K2 [30], (ii) Log-likelihood (LL) [31], (iii) Minimum Description Length (MDL) [32], and (iv) Mutual information (MI) in Eq.(4). Then, we evaluate PLC-Sleuth’s intrusion detection/localization under two attack scenarios:

- **Attack-1:** a control command injection attack implemented by injecting time-varied values to the PIO table (including those of only slight deviations from normal commands) and tampering the control algorithm parameters.
- **Attack-2:** a cooperative stealthy attack atop of Attack-1 by replaying normal sensor measurements during attack.

5.2 Evaluation with SEDS

We first evaluate PLC-Sleuth using SEDS.

5.2.1 Constructing SEDS’s Control Graph

We first examine if PLC-Sleuth is able to accurately construct SEDS’s control graph.

Data Collection. We logged a 9.8h operation of SEDS using its SCADA system, during which a total number of 17 variables are collected at 2Hz, including 3 setpoint variables (i.e.,

$|S| = 3$), 11 sensor variables (i.e., $|Y| = 11$), and 3 command variables (i.e., $|U| = 3$). We then use these logs to evaluate PLC-Sleuth’s construction of the control graph. Note SEDS has three decoupled single feedback loops, as shown in Fig. 7.

Accuracy of Graph Construction. With a training data of 2h, Table 1 compares the weights of $e_{y_j}^{u_k}$ s (i.e., $w_{y_j}^{u_k}$ s) of SEDS obtained with PLC-Sleuth and when using K2, LL, MDL, and MI as the weighting metric, showing:

- PLC-Sleuth identifies SEDS’s command edges $e_{y_j}^{u_k}$ s accurately and without repeated edges;
- K2 and MDL fail to identify the command edges correctly, due to the falsely matched edges of $\{e_{y_1}^{u_3}, e_{y_3}^{u_1}\}$ and $\{e_{y_1}^{u_2}, e_{y_2}^{u_1}\}$, respectively;
- Although LL and MI also match sensor nodes to their corresponding command nodes successfully, they cannot accurately characterize the correlation strength between nodes u_1 and y_1 , as evident by their results of $w_{y_1}^{u_1} < w_{y_3}^{u_1}$.

We have further evaluated PLC-Sleuth with varying volumes of training data, as plotted in Fig. 11, where the accuracy is averaged over 1,000 runs by randomly selecting the start time of training sequences from the 9.8h system logs. PLC-Sleuth’s accuracy of graph construction outperforms all other four weighting schemes, and increases with a longer training data — achieving 100% when being trained with a system log of 0.5h.

5.2.2 Intrusion Detection/Localization with SEDS

We next evaluate PLC-Sleuth’s intrusion detection/localization under two attack scenarios. The command injection attacks are launched in two ways, i.e., replacing output command series with designed attack vector in PIO table (e.g., replacing $u_2 = [0.2757, 0.2762, 0.2766]$ with $u_2^a = [0.2967, 0.2968, 0.2969]$), and tampering parameters of control rules (e.g., changing K_p in the rule of $\Delta u_2 = -K_p(\Delta x_2 + 0.17 \times x_2)$ from $K_p = 0.01$ to $K_p^a = 0.1$). Each of the control loop is attacked 10 times, i.e., a total number of $10 \times 3 = 30$ attacks are mounted to SEDS. Besides, by replaying the corresponding sensor’s normal logs, we mount another $10 \times 3 = 30$ cooperative stealthy attacks.

Detection/Localization Accuracy. Table 2 lists the evaluation results, showing that PLC-Sleuth achieves an average of 98.33%/0.85% true/false positive (TP/FP) alarm rate, and localizes the forged command with 93.22% (i.e., 55 out of 59) positive predictive value (PPV).

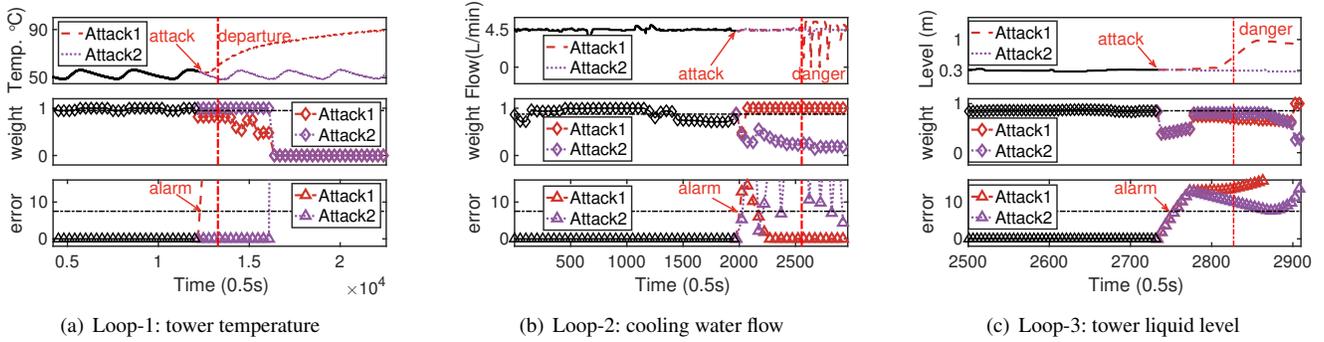


Figure 12: Detecting the two attacks mounted at SEDS's three control loops.

Table 2: Evaluating PLC-Sleuth's intrusion detection/localization using SEDS.

Loop No.	Attack-1			Attack-2		
	TP	FP	PPV	TP	FP	PPV
#1	100%	1.3‰	90%	90%	1.3‰	66.7%
#2	100%	0.35‰	100%	100%	0.35‰	100%
#3	100%	0.26‰	100%	100%	0.26‰	100%

¹ Detection parameters: $T = 500$ for Loop-#1 and Loop-#2, $T = 4000$ for Loop-#3; $\mu = 7.5$; $\tau = 5$.

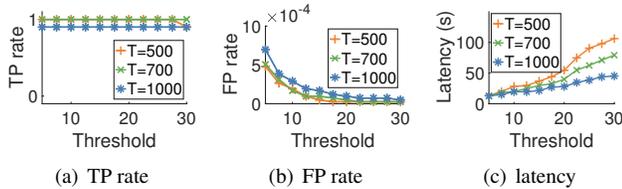


Figure 13: Applying PLC-Sleuth to SEDS with varying detection window and threshold.

- **Attack-1.** The injected forged command incurs instability to SEDS's distillation process, as shown with the red dashed line of the upper subplots in Fig. 12. PLC-Sleuth detects the weight changes of the monitored edges immediately after the attack, and thus triggers alarms. Among the 10 command injection attacks against the tower level loop, Figs. 13(a) and 13(b) plot PLC-Sleuth's detection results with various configurations of detection window (T) and threshold (μ). Increase of the threshold value hardly affects the TP rate, but reduces FP alarms noticeably. However, a larger detection window causes a relatively lower TP rate due to its insensitivity to parameter changing attacks.
- **Attack-2.** The cooperatively replayed sensor logs hide attacks from the operators, as shown with the purple dotted line of the upper subplots in Fig. 12. However, the monitored weight deviates from its normal value noticeably. Note that one missed TP alarm occurs to **Loop-#1**, (i.e., the temperature loop), which has a long control cycle of 1,800s, causing insensitive variation to variables' correlation. The PPV of localization for **Loop-#1** is relatively low (when compared to Loop-2 and 3) for the same reason.

Detection Latency. We have also examined PLC-Sleuth's

latency in detecting the PLC intrusions. Although a delay exists for an attack to physically disturb the system, the weight of control graph changes instantly when the attack is launched, as shown in the middle subplots of Fig. 12. As a result, PLC-Sleuth detects these attacks with a short latency, e.g., {50, 12.5, 10.5}s for attacks in Fig. 12(a), 12(b), and 12(c), respectively. Besides, Fig. 13(c) shows that a larger detection threshold increases the detection latency. On the other hand, the detection latency decreases with a larger detection window T , because a larger time window facilitates PLC-Sleuth capturing the control invariant (i.e., weight $w_{y_j}^{u_k}$) more reliably.

5.3 Evaluation with TE Process

To further evaluate PLC-Sleuth when being deployed at a large-scale control system, we implement PLC-Sleuth on a realistically simulated TE process (see Fig. 21 in Appendix) [33]. The TE process contains 12 setpoint variables in S , 41 measurement variables in Y , 12 terminal command variables in U , and 14 transfer variables in R . These variables together form 17 feedback loops, 16 of which form 7 multiple feedback loops.

5.3.1 Constructing TE's Control Graph

Again, we first examine PLC-Sleuth's accuracy of constructing TE's control graph.

Data collection. Simulating the TE process with Matlab, we obtained a 72h training data logged at 1.8Hz. The (correctly) constructed control graph is shown in Fig. 14, in which a total number of 45 edges need to be identified.

Accuracy of Graph Construction. We have evaluated PLC-Sleuth with various volumes of training data in different running periods. Fig. 15 plots the results averaged over 1,000 runs. PLC-Sleuth's graph construction achieves a high accuracy of 95.76% even when being trained with only a short trace of 2h, which increases further with a larger volume of training data. The average accuracy reaches 98.11% with a 20h training data, i.e., PLC-Sleuth accurately identifies almost all of TE's 45 edges. Note that the construction errors mainly occur at transfer variables, because of the similarities

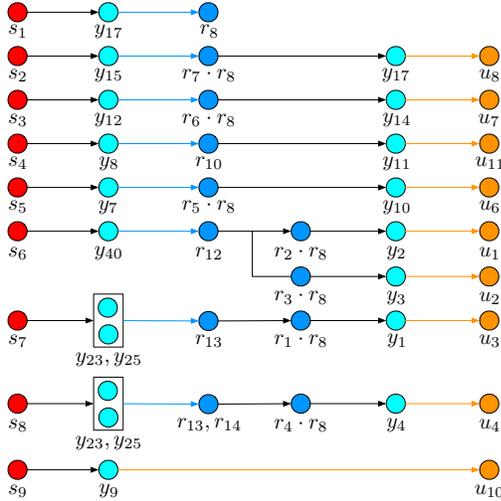


Figure 14: Control graph of the TE process.

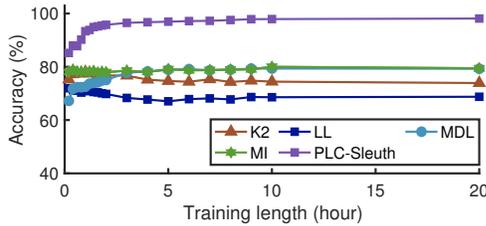


Figure 15: Constructing the TE process’s control graph.

between transfer nodes and transmit nodes (e.g., mismatching r_2 to y_{40} in Fig. 14). Fig. 15 also plots the results of control graph construction when using different metrics as $w_{y_j}^{u_k}$, showing that PLC-Sleuth achieves the highest construction accuracy.

5.3.2 Intrusion Detection/Localization with TE

We next examine PLC-Sleuth’s intrusion detection/localization with two attack scenarios in TE. Each of the 17 loops is attacked for 10 times in both scenarios. PLC-Sleuth detects attacks with 100% TP alarm rate, without any FP alarms. For the 340 alarms, PLC-Sleuth localizes the forged command with a PPV of 96.76% (i.e., 329 out of 340). Detailed detecting/localizing results of the $17 \times 10 \times 2 = 340$ attacks are listed in Table 3 of Appendix.

- **Attack-1.** The forged time series, which are generated by (i) adding random noise to normal output commands, and (ii) mixing constant values with random white noises to hide the attack, are injected to each of the 17 control loops. Fig. 16 plots the weight $w_{y_j}^{u_k}$ under the four example attacks. We can see that the weights of all the edges $e_{y_j}^{u_k}$ corresponding to the compromised control loops change due to the attack, making them detectable by PLC-Sleuth. Figs. 23(a) and 23(b) in Appendix present a visualization of the weight change. We further evaluate the impacts of detection win-

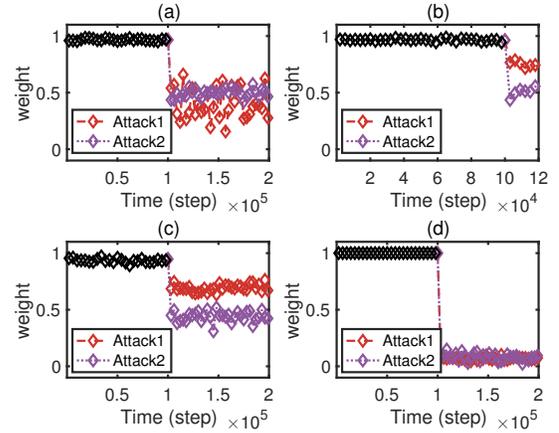


Figure 16: Command weights $W_{y_j}^{u_k}$ of four control loops under two attacks: (a) reactor temperature; (b) reactor level; (c) reactor pressure; (d) product quality.

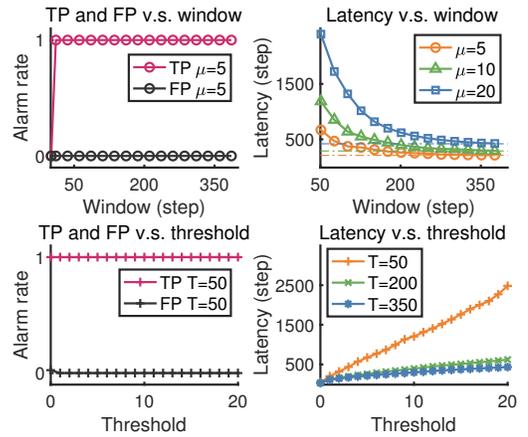


Figure 17: Impact of the detection window (T) and threshold (μ) on PLC-Sleuth’s attack detection for the reactor level control in TE.

dow and threshold using the 10 attacks against the TE’s reactor level control, as plotted in Fig. 17, showing that detection window and threshold have limited impacts on the TP/FP rate, but have significant influence on the detection latency — the detection latency is inversely proportional to the window size. This is because the detection data with a long period of logs is able to capture better the normal command weight and thus identify the anomalies.

- **Attack-2.** By injecting the forged command u^a into each control loop and replacing the sensor measurements with their historical normal records, another $17 \times 10 = 170$ attacks are performed. These attacks have similar impacts on the TE process as with *Attack-1*. The monitored weight $w_{y_j}^{u_k}$ changes significantly regardless of the replayed sensor measurements, as plotted in Fig. 16 and visualized in Figs. 23(a) and 23(c) in Appendix. Note PLC-Sleuth detects Attack-2 against the reactor pressure control loop with $\{100\%, 0\%\}$ true/false positives, as long as the deviation from normal

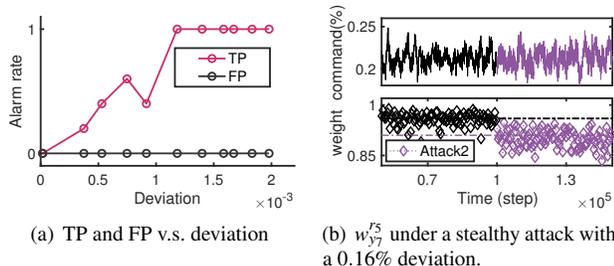


Figure 18: PLC-Sleuth’s sensitivity to deviations from the normal commands for the reactor pressure control in TE.

commands (defined as $\sum |u^a(k) - u(k)| / \sum u(k) \times 100\%$) is larger than 0.12%, as shown in Fig. 18(a). Fig. 18(b) visualizes the weight change with a 0.16% deviation.

5.3.3 Tolerance to Inaccurate Control Graph

We examine PLC-Sleuth’s tolerance to the few cases of inaccurately constructed control graph. With PLC-Sleuth, falsely matched edges mainly occur at control loops containing transfer variables, which we call inner-loop fault, such as the edge connecting node y_{40} to r_2 in Fig. 14. For the control graph constructed by other learning metrics, falsely matched edges also include the pairs of nodes from different control loops, which we call the inter-loop fault, such as edge connecting node y_{11} to u_8 in Fig. 14. Fig. 19 shows detection results under the two attack scenarios, by using the control graph with two different construction errors. The control graph with inner-loop construction faults still detects the attack, although with a higher FP rate; the one with inter-loop faults, however, is not applicable anymore.

6 Discussion

Stealthy sensor attack. Orthogonal to the targeted cooperative stealthy attacks of PLC-Sleuth, the stealthy sensor attacks have been investigated extensively [34–36], which can be detected effectively using process invariants defined using the correlations among sensors [28, 37–39].

Sophisticated Evasive Attacks. To evade PLC-Sleuth, sophisticated attackers may carefully forge sensor logs to make them consistent with expected correlations with the forged commands. However, it can be challenging to launch such an attack in practice. First, the attacker would need deep knowledge of not only pairs of correlated variables in the same control loop but also the control algorithm that governs their detailed relationships. Second, control loops are interdependent by the laws of physics, thus it is hard for the attacker to mimic sensors’ behavior, as he needs to jointly model all the relevant control loops. These limitations make the further characterization of evasive cooperative stealthy attacks an interesting topic for future research.

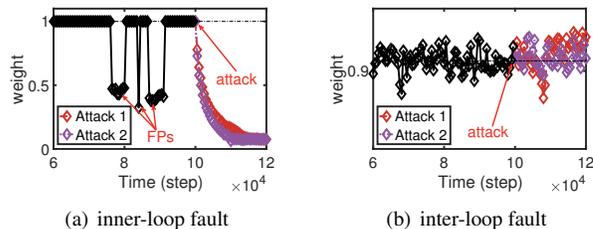


Figure 19: Weights of the two types of the false matched edges under the two attacks scenarios.

7 Related work

PLC attacks. Systematic attacks against PLC have been reported. PLC firmware, providing an interface between user program and PLC input/output modules, has been attacked in [8, 11, 40]. The compromised firmware separates legitimate control rules from the physical process. A “ladder logic bombs”, instantiated in IEC 61131-3 languages, was designed against a PLC user program to activate malicious behaviour under pre-defined conditions [41]. Besides, a dynamic payload generation against the target physical process has been proposed in [42] and enhanced by [6]. They generated the malicious payloads automatically.

Detecting PLC attacks. Against PLCs’ attacks, many detection methods have been proposed based on the following methodologies: program analysis and system modeling. Program analysis based detections [43–46] monitor the control flow integrity (CFI) when controller is processing. However, the CFI checking is time-consuming, which renders it unsuitable for real-time detection in many situations [47]. System modeling methods that characterize variables’ static invariants have been investigated [37, 48–52]. Further efforts have also been made to build dynamic models/invariants [39, 53–55]. In comparison, PLC-Sleuth also mines dynamic invariants but does so using significantly less a priori knowledge.

8 Conclusion

We have proposed PLC-Sleuth, a novel attack detection/localization scheme grounded on a PLC’s control graph \mathcal{G} . The control graph describes a control invariant of PLC, with the inherent and essential characteristics of control loops. Using the constructed control graph, PLC-Sleuth flags and localizes attacks when weights in \mathcal{G} deviates from the norm. Evaluation results using SEDS and TE process show that PLC-Sleuth can construct control graph with high accuracy (100% with a log of 0.5h for SEDS and 98.11% with a log of 20h for TE process). PLC-Sleuth achieves detection true/false positives of {98.33%, 0.85%} for SEDS, and of {100%, 0%} for TE process. In terms of attack localization, PLC-Sleuth localizes the forged command with a {93.22, 96.76}% accuracy for SEDS and TE process, respectively.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0803501, the National Natural Science Foundation of China under Grant 61833015, U1911401, the SUTD-ZJU IDEA grant number SUTD-ZJU (VP) 201805, the Singapore MOE T1 grant number SUTDT12017004, and a startup grant of University of Colorado Denver.

References

- [1] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to industrial control systems (ICS) security. *NIST special publication*, 800(82):16–16, 2011.
- [2] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 328–335, 2019.
- [3] Ariana Mirian, Zane Ma, David Adrian, Matthew Tischer, Thasphon Chuenchujit, Tim Yardley, Robin Berthier, Joshua Mason, Zakir Durumeric, J. Alex Halderman, and Michael Bailey. An Internet-wide view of ICS devices. In *14th Annual Conference on Privacy, Security and Trust (PST)*, pages 96–103, 2016.
- [4] Qiang Li, Xuan Feng, Haining Wang, and Limin Sun. Understanding the usage of industrial control system devices on the internet. *IEEE Internet of Things Journal*, 5(3):2178–2189, 2018.
- [5] Symantec Security Response. Dragonfly: Western energy sector targeted by sophisticated attack group. <https://www.symantec.com/blogs/threat-intelligence/dragonfly-energy-sector-cyber-attacks>, 2017.
- [6] Anastasis Keliris and Michail Maniatakos. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.
- [7] Stephen McLaughlin and Patrick McDaniel. SABOT: specification-based payload generation for programmable logic controllers. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 439–449, 2012.
- [8] Zachry Basnight, Jonathan Butts, Juan Lopez, and Thomas Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76 – 84, 2013.
- [9] Symantec Security Response. After Triton, Will the Industrial Threat Landscape Ever be the Same? <https://www.symantec.com/blogs/feature-stories/after-triton-will-industrial-threat-landscape-ever-be-same>, 2018.
- [10] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier. *White paper, Symantec Corp., Security Response.*, 5(6):29, 2011.
- [11] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *Network and Distributed Systems Security (NDSS) Symposium*, 2017.
- [12] Gene F Franklin, J David Powell, and Abbas Emami-Naeini. *Feedback control of dynamic systems*. Pearson London, 2015.
- [13] Gary A Dunning. *Introduction to programmable logic controllers*. Cengage Learning, 2005.
- [14] Henry Z Kister. What caused tower malfunctions in the last 50 years? *Chemical Engineering Research and Design*, 81(1):5–26, 2003.
- [15] Richard Bellman, Irving Glicksberg, and Oliver Gross. On the “bang-bang” control problem. *Quarterly of Applied Mathematics*, 14(1):11–18, 1956.
- [16] Karl Johan Åström, Tore Hägglund, and Karl J Astrom. *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society, 2006.
- [17] Hans Berger. *Automating with STEP7 in STL and SCL: programmable controllers Simatic S7-300/400*. Publicis Publishing, 2012.
- [18] Pedro Albertos and Sala Antonio. *Multivariable control systems: an engineering approach*. Springer Science & Business Media, 2003.
- [19] Yilin Mo and Bruno Sinopoli. Secure control against replay attacks. In *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 911–918, 2009.
- [20] Xue-Wen Chen, Gopalakrishna Anantha, and Xiaotong Lin. Improving Bayesian network structure learning with mutual information-based node ordering in the K2 algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):628–640, 2008.
- [21] Xiannian Fan and Changhe Yuan. An improved lower bound for bayesian network structure learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3526–3532, 2015.

- [22] Pengfei Sun, Luis Garcia, and Saman Zonouz. Tell Me More Than Just Assembly! Reversing Cyber-physical Execution Semantics of Embedded IoT Controller Software Binaries. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 349–361, 2019.
- [23] Ralf Steuer, Jürgen Kurths, Carsten O Daub, Janko Weise, and Joachim Selbig. The mutual information: detecting and evaluating dependencies between variables. *Bioinformatics*, 18(suppl_2):S231–S240, 2002.
- [24] Thomas McAviney and Raymond Mulley. *Control System Documentation: Applying Symbols and Identification*. ISA, 2004.
- [25] Sigurd Skogestad and Ian Postlethwaite. *Multivariable feedback control: analysis and design*. Wiley New York, 2007.
- [26] Emerson report. Fisher 4320 Wireless Position Monitor. <https://www.emerson.com/en-us/catalog/fisher-4320>, 2009.
- [27] SIEMENS report. SIEMENS SIPART PS2 (6DR5...) Electropneumatic positioners. http://www.lesman.com/unleashd/catalog/accessor/Siemens_SIPART-PS2/Siemens-SIPART-PS2-man-A5E03436620-AB-2017-01.pdf, 2017.
- [28] David I Urbina, Jairo A Giraldo, Alvaro A Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1092–1105, 2016.
- [29] J.J. Downs and E.F. Vogel. A plant-wide industrial process control problem. *Computers & Chemical Engineering*, 17(3):245–255, 1993.
- [30] Gregory F Cooper and Edward Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347, 1992.
- [31] Remco Ronaldus Bouckaert. *Bayesian belief networks: from construction to inference*. PhD thesis, 1995.
- [32] Wai Lam and Fahiem Bacchus. Learning Bayesian belief networks: An approach based on the MDL principle. *Computational intelligence*, 10(3):269–293, 1994.
- [33] N. Lawrence Ricker. Decentralized control of the Tennessee Eastman Challenge Process. *Journal of Process Control*, 6(4):205–221, 1996.
- [34] Yao Liu, Peng Ning, and Michael K. Reiter. False Data Injection Attacks against State Estimation in Electric Power Grids. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, page 21–32, 2009.
- [35] Saurabh Amin, Xavier Litrico, Shankar Sastry, and Alexandre M Bayen. Cyber security of water SCADA systems—Part I: Analysis and experimentation of stealthy deception attacks. *IEEE Transactions on Control Systems Technology*, 21(5):1963–1970, 2012.
- [36] Pritam Dash, Mehdi Karimibiuki, and Karthik Pattabiraman. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 660–672, 2019.
- [37] Yong Wang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Haopei Wang, and Guofei Gu. Srid: State relation based intrusion detection for false data injection attacks in scada. In *European Symposium on Research in Computer Security*, pages 401–418, 2014.
- [38] Wissam Aoudi, Mikel Iturbe, and Magnus Almgren. Truth Will Out: Departure-Based Process-Level Detection of Stealthy Attacks on Control Systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 817–831, 2018.
- [39] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Lin Zhiqiang. SAVIOR: Securing Autonomous Vehicles with Robust Physical Invariants. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [40] Ali Abbasi and Majid Hashemi. Ghost in the PLC Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack. In *Black Hat Europe.*, 2016.
- [41] Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. On ladder logic bombs in industrial control systems. In *Computer Security*, pages 110–126, 2018.
- [42] Stephen McLaughlin. On dynamic malware payloads aimed at programmable logic controllers. In *Proceedings of the 6th USENIX conference on Hot topics in security*, 2011.
- [43] Lucille McMinn and Jonathan Butts. A firmware verification tool for programmable logic controllers. In *International Conference on Critical Infrastructure Protection*, pages 59–69, 2012.
- [44] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM*

International Conference on Automated Software Engineering, pages 338–341, 2012.

- [45] Stephen McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. A Trusted Safety Verifier for Process Controller Code. In *Network and Distributed Systems Security (NDSS) Symposium*, 2014.
- [46] Saman Zonouz, Julian Rrushi, and Stephen McLaughlin. Detecting industrial control malware using automated PLC code analytics. *IEEE Security Privacy*, 12(6):40–47, 2014.
- [47] Irfan Ahmed, Sebastian Obermeier, Sneha Sudhakaran, and Vassil Roussev. Programmable Logic Controller Forensics. *IEEE Security Privacy*, 15(6):18–24, 2017.
- [48] Robert Mitchell and Ray Chen. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing*, 12(1):16–30, 2014.
- [49] Khurum Nazir Junejo and Jonathan Goh. Behaviour-based attack detection and classification in cyber physical systems using machine learning. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, pages 34–43, 2016.
- [50] Yuqi Chen, Christopher M. Poskitt, and Jun Sun. Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 240–252, 2018.
- [51] Feng Cheng, Palle Venkata, Reddy, Mathur Aditya, and Chana Deeph. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.
- [52] Mu Zhang, James Moyne, Z Morley Mao, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, et al. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *2019 IEEE Symposium on Security and Privacy (S&P)*, pages 522–538, 2019.
- [53] Pinyao Guo, Hunmin Kim, Nurali Virani, Jun Xu, Minghui Zhu, and Peng Liu. RoboADS: Anomaly detection against sensor and actuator misbehaviors in mobile robots. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 574–585, 2018.
- [54] Hongjun Choi, Wen-Chuan Lee, Youssa Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 801–816, 2018.
- [55] Hamid Reza Ghaeini, Matthew Chan, Raad Bahmani, Ferdinand Brasser, Luis Garcia, Jianying Zhou, Ahmad-Reza Sadeghi, Nils Ole Tippenhauer, and Saman Zonouz. PAtt: Physics-based Attestation of Control Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 165–180, 2019.

Appendix

Patterns of SCADA logs

SCADA logs present obvious three kinds of patterns, which correspond to the 3 defined node sets:

- Setpoint set S . Setpoint variables are of constant values, shown in Figs. 20(a)-20(c);
- Sensor set Y . Sensor readings are of consistent and small vibrations, shown in Figs. 20(d)-20(n).
- Command set U . Control commands are of continuous/smooth values, as shown in Figs. 20(o)-20(q).

Evaluating PLC-Sleuth on TE

Tennessee Eastman (TE) process, as shown in Fig. 21, is widely used for evaluating threats and protection methods designed for cyber physical systems.

We deploy the command injection attack and the cooperative stealthy attack on the 17 control loops of TE. Fig. 22 shows the impacts of four example attacks.

As a typical control system, the control commands in TE have strong correlations with the concomitant sensor readings, shown as the black line in Fig. 23. This correlations can be quantified as command weights $w_{y_j}^{u_k}$ s. However, the two scenarios of attacks both damage the correlations and cause command weights' variation, shown as the red and purple line in Fig. 23.

By utilizing the variation of command weights, PLC-Sleuth detects attacks with $\{100\%/0\}$ true/false positives, and localizes the compromised control loops with 96.76% accuracy. Details of the detection/localization results are listed in Table 3.

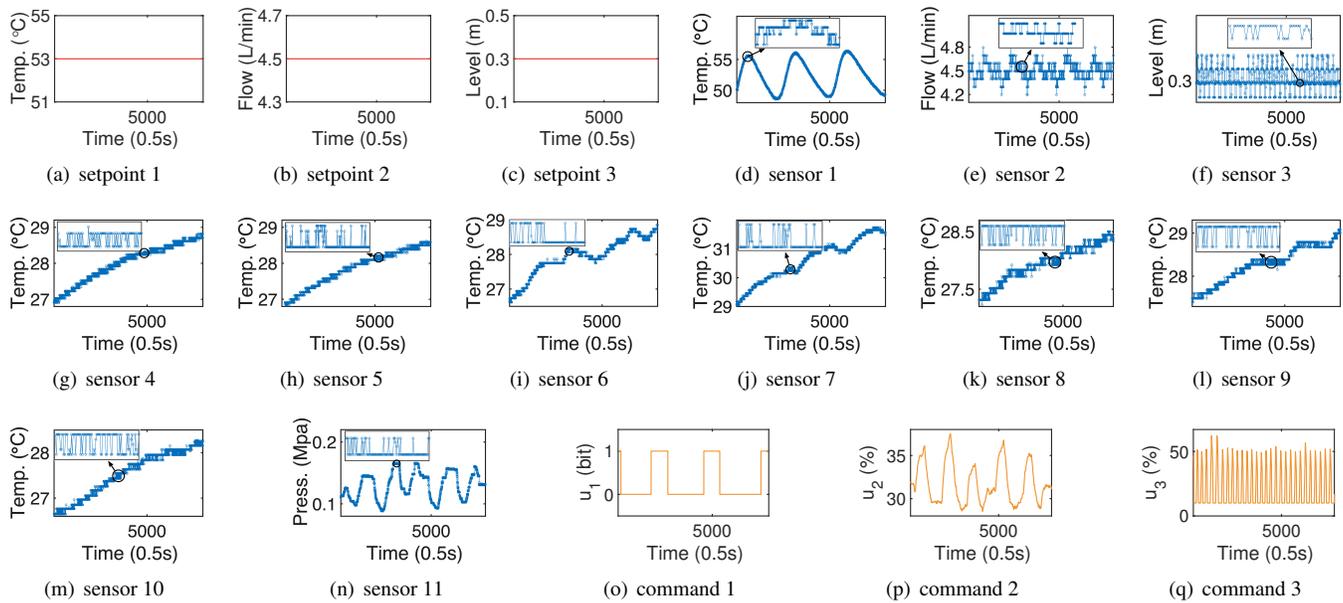


Figure 20: Different patterns of setpoints, sensors, commands variables in SEDS.

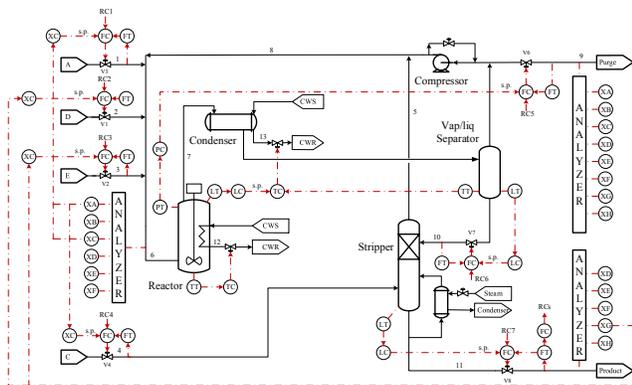


Figure 21: The Tennessee Eastman (TE) control process [33].

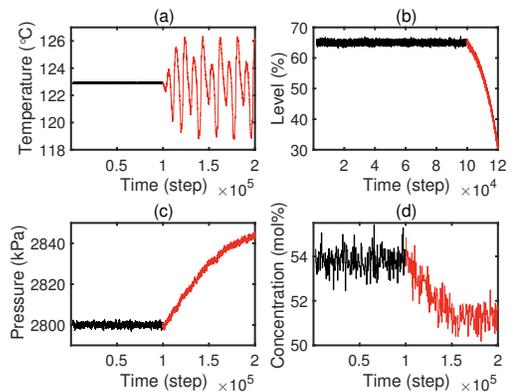


Figure 22: System evolution under command injection attacks: (a) unstable reactor temperature; (b) stopped reactor level; (c) dangerous reactor pressure; (d) degraded product quality.

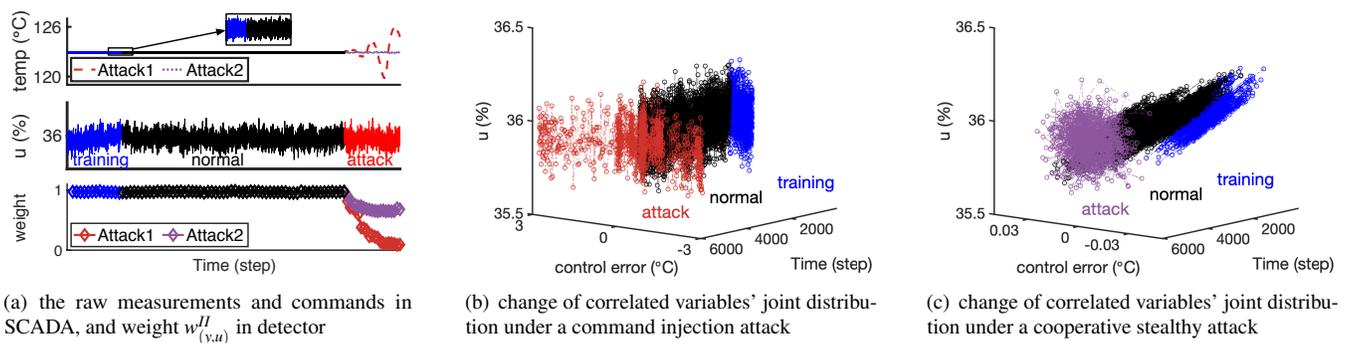


Figure 23: A visualization of abnormal weight in TE's control graph under the two attacks against reactor temperature control.

Table 3: Evaluating PLC-Sleuth’s attack detection and localization with the TE process.

Compromised Control loop Number	Controlled variable [◊]	Command variable [◊]	Detection				Localization	
			command injection		stealthy attack		command injection	stealthy attack
			TP rate	FP rate	TP rate	FP rate		
1	A feed rate	$xmv(3)$	100%	0	100%	0	100%	100%
2	D feed rate	$xmv(1)$	100%	0	100%	0	100%	100%
3	E feed rate	$xmv(2)$	100%	0	100%	0	100%	100%
4	C feed rate	$xmv(4)$	100%	0	100%	0	100%	50%
5	Purge rate	$xmv(6)$	100%	0	100%	0	100%	100%
6	Sep.liq.rate	$xmv(7)$	100%	0	100%	0	100%	100%
7	Strip.liq.rate	$xmv(8)$	100%	0	100%	0	100%	100%
8	Production rate	F_p	100%	0	100%	0	80%	80%
9	Strip.liq.level	Ratio in loop 7	100%	0	100%	0	100%	100%
10	Sep.liq.level	Ratio in loop 6	100%	0	100%	0	100%	100%
11	Reac.liq.level	Setpoint of loop 17	100%	0	100%	0	100%	100%
12	Reac.pres	Ratio in loop 5	100%	0	100%	0	100%	100%
13	Mol%G in stream 11	E_{adj}	100%	0	100%	0	100%	100%
14	y_A	Ratio in loop 1, r_1	100%	0	100%	0	100%	90%
15	y_{AC}	Sum of $r_1 + r_4$	100%	0	100%	0	100%	90%
16	Reac.temp	$xmv(10)$	100%	0	100%	0	100%	100%
17	Sep.temp	$xmv(11)$	100%	0	100%	0	100%	100%

[◊] Controlled variables and command variables are defined in [29,33].

Software-based Realtime Recovery from Sensor Attacks on Robotic Vehicles

Hongjun Choi¹, Sayali Kate¹, Yousra Aafer², Xiangyu Zhang¹, and Dongyan Xu¹

¹Purdue University

²University of Waterloo

¹{choi293, skate, xyzhang, dxu}@purdue.edu

²yaafer@uwaterloo.ca

Abstract

We present a novel technique to recover robotic vehicles (RVs) from various sensor attacks with so-called *software sensors*. Specifically, our technique builds a predictive state-space model based on the generic system identification technique. Sensor measurement prediction based on the state-space model runs as a software backup of the corresponding physical sensor. When physical sensors are under attacks, the corresponding software sensors can isolate and recover the compromised sensors individually to prevent further damage. We apply our prototype to various sensor attacks on six RV systems, including a real quadrotor and a rover. Our evaluation results demonstrate that our technique can practically and safely recover the vehicle from various attacks on multiple sensors under different maneuvers, preventing crashes.

1 Introduction

Robotic Vehicles (RVs) are complex cyber-physical systems (CPS) that continuously change their physical states based on sensor measurements. Specifically, various sensors monitor the current system's physical states and the environment. Based on the measurements, the control components generate actuation signals to control the vehicle for stable operations according to the planned behaviors. RVs, such as drones, ground rovers, and underwater robots [2, 6, 50], utilize multiple sensors of different types. For example, a gyroscope sensor measures angular velocities, an accelerometer measures linear accelerations, a GPS provides geographic position information, and a barometer measures the pressure outside the vehicle which is used for altitude calculation. Unlike the traditional cyber attacks, attackers aiming at RVs can compromise sensor readings through external and physical channels. Since RVs operate based on sensors, the security of RV sensors has become a primary requirement and challenge.

Along with the wide deployment of safety-critical RVs, many physical sensor attacks have been reported recently. For instance, GPS spoofing [51, 54] is a typical physical sensor

attack to deceive GPS receiver by injecting incorrect GPS signals. Gyroscopic sensor attack on UAV systems through sound noises [49] can disrupt attitude measurements and lead to crashes. Attackers can manipulate the measurements of MEMS accelerometers via analog acoustic signal injection in a controlled manner [52]. In optical sensor spoofing [8], attackers can acquire an implicit control channel by deceiving the optical flow sensor of a UAV with a physically altering ground plane. Attackers in [47] corrupt automobile's Anti-lock Braking System (ABS) by injecting magnetic fields to wheel speed sensors. In [42], researchers presented remote attacks on camera and LiDAR systems in a self-driving car by introducing false signals with a cheap commodity hardware. These physical sensor attacks pose new challenges because the traditional techniques to protect software are deficient.

To defend the external attacks, many methods have been published recently [5, 22, 26, 36, 37, 57]. However, they only focus on attack detection rather than attack resilience, which is not a complete solution. A canonical counter-measure for attack recovery is to leverage hardware redundancy [29], where critical components are multiplied to provide attack resilience. For instance, triple module redundancy (TMR) uses three sensors to measure the same physical properties and produces a single output by majority-voting or weighted average. This approach requires additional cost to deploy and sustain the redundant hardware. Additionally, an adversary can still attack multiple sensors as all these sensors are exposed to the same compromised physical environment.

We propose a novel *software sensor* recovery technique for multi-sensor RVs to be resilient to *physical sensor attack*. Instead of using duplicated hardware, our approach uses so-called *software sensors* as the backup of the corresponding physical sensors. Our method can recover when multiple sensors of the same kind or different kinds are under attack. Unlike the transitional physical control systems, the emergence of computationally powerful CPS allows new opportunities to deploy more complex software-based control and recovery components. With these advantages, a software sensor continuously computes and predicts the reading of the corre-

sponding physical sensor. When an attack is detected on some physical sensor(s), the corresponding software sensor(s) allow to isolate and replace the compromised sensors, and recover the system from corrupted internal states to prevent serious attack consequences (e.g., crashes and physical damages). Since our approach is purely software-based, not requiring any additional hardware (e.g., HW duplication and mechanical shielding), it can be deployed not only at design time but also to patch existing systems (e.g., legacy systems).

Specifically, our technique builds a precise state-space model of the vehicle that allows us to predict its physical states (i.e., expected physical behaviors). The model is largely determined by the gravity, control algorithms used, and the physical characteristics of the vehicle (e.g., motor specification, weight and frame shape). We then construct a set of *software sensors*, one for each physical sensor, by transforming the predicted physical states (i.e. the model output) into the appropriate sensor readings using the mathematical conversion equations. In practice, the predicted sensor readings tend to deviate from the real sensor readings due to various reasons. Therefore, to compensate for the intrinsic errors (*conversion, model, and external errors*), we further develop a number of error correction techniques.

Software sensor readings and physical sensor readings are continuously monitored and compared. In normal operations, both readings are almost identical. Substantial discrepancies indicate that the corresponding physical sensor is under attack. The compromised sensor is hence replaced with its software version. Note that software sensors do not interact with the (compromised) physical environment, thus allowing the vehicle to continue normal operation (for a certain time duration) in the presence of the attacks.

Contribution. Our contributions are summarized as follows.

- We propose a novel software-based technique, *software sensors*, for recovery from various physical and external sensor attacks. This is the first work on the sensor recovery for RVs with the software sensors.
- We address a number of prominent challenges, including how to generate software sensors using system identification; how to recover from individual sensor failures; and how to improve software sensor accuracy considering external disturbances for practical usage.
- We conduct a set of comprehensive experiments on multiple RVs, including simulated RVs and two real ones (a quadrotor and a rover), using various kinds of attacks on one or multiple sensors. The results show that with low overhead, our framework can successfully recover from all the attacks considered for all the RVs, effectively preventing physical damage to the subject vehicles.

Adversary Model. We target physical sensor attacks that maliciously corrupt sensor signals through *external* channels. Additionally, we assume that the attacker can compromise multiple sensors at the same time with different attack techniques,

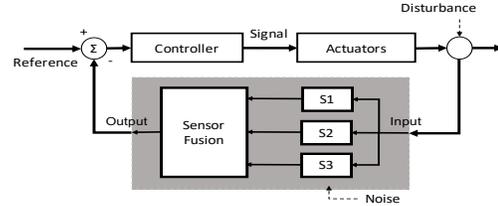


Figure 1: Feedback control loop with sensor redundancy

resulting in disrupted sensor readings. The state-of-the-art attacks (Section 6) can inject both noise or sensor values in a controlled manner. However, we assume that the attacker does not have access to the internal states of subject RV, such as the true sensor readings and the intended navigation plan, thus cannot generate constant deviation (smaller than any pre-defined threshold). We argue that this is reasonable for the following. (1) We target physical attack channels, for example, acoustic noise (to attack inertia sensors), under which achieving fine-grained manipulation is difficult. (2) To ensure the applied error is consistently smaller than the detection threshold, the attacker needs to have precise estimation of the RV internal sensor readings, in the 400Hz (2.5ms) time unit, which is practically hard by observing external behaviors. Note that while the attacker may use external observation and modeling to estimate RV internal sensor readings when the RV operates normally and has a predictable navigation plan, such estimation becomes infeasible when the navigation plan is not predictable and the RV’s internal states have been corrupted by the attack itself. Without precise estimation, the injected error may exceed the threshold and will be detected by our technique (see Section 4.3 for an example).

We do not consider traditional attacks on software or firmware in the cyber domain since those attacks can be effectively handled by existing software security techniques [7, 43]. Thus, we assume that our recovery framework – running as part of the control program – is safe against cyber attack vectors.

2 Motivation and Background

In this section, we first introduce control loop with hardware redundancy as background. Later, we use an example to illustrate the physical sensor attack and recovery with the proposed approach. This example simulates a sensor spoofing attack on a real quadrotor by artificially inserting malicious signal data.

2.1 Background

A common control mechanism in RVs shown in Figure 1 is the feedback-loop control which takes system outputs (i.e., current physical state) as the input in the loop. The controller adjusts its control signal to make the vehicle reach the reference state over the loop. Most RVs utilize multi-sensor measurements to obtain a more accurate view of physical

state since a single sensor cannot provide reliable data in a real environment (due to sensor noises, possible sensor failure, etc.). In quadrotors, multiple redundant or heterogeneous sensors (e.g., gyroscopes, accelerometers, and GPS) enable the controller to recognize the current physical state and the environment, and then accordingly control motor signals for a stable flight.

Sensor fusion [4] is a very common practice in control engineering. The technique combines multi-sensor data to produce enhanced results. Figure 1 shows typical sensor fusion with the triple modular redundancy (TMR). A single physical property is measured by multiple sensors, and a fusion algorithm combines the redundant information to generate a single output with high accuracy in a competitive way (e.g., voting) or a complementary way (e.g., weighted average). Sensor fusion is not limited to the same type of sensors. Complex sensor fusion algorithms (e.g., extended Kalman filter) often utilize heterogeneous sensor data to reduce uncertainty and produce more accurate measurements. Although sensor fusion can improve accuracy and tolerate failures of a subset of sensors (of a specific kind), it is not effective for physical attacks. For example, the sensor fusion with TMR utilizes the majority voting technique in which, if any one out of the three sensors is compromised or faulty, the other two sensors can identify and mask the faulty one. However, if two sensors (the majority of the sensors) are compromised at the same time, it is difficult to identify which sensors are problematic, which is the Byzantine agreement problem [31]. In case of sensor fusion with the weighted average technique, any single compromised sensor can significantly degrade control performance.

2.2 Motivating Example



Figure 2: Sequential snapshots from the video of the gyroscope sensors attack (the full video is available at [11]).

Sensor spoofing attack [8, 30, 41, 47, 49, 51–54] is a popular physical attack on RVs. The adversaries maliciously disrupt sensor measurements by perturbing the physical environment or directly compromising sensor internals with physical means. In our example, we use a real commodity quadrotor, 3DR Solo. The vehicle is equipped with three Inertia Measurement Units (IMU), each including a gyroscope, an accelerometer, and a magnetometer. Among the different sensors, we aim to disrupt multiple gyroscope readings with a simulated acoustic sensor attack, leading to a physical crash. In particular, the attacker intentionally injects acoustic noises at the resonant frequency of the gyroscopes, causing the gyroscopes to generate erroneous angular rates. Here, we assume

that the attacker cannot access the internals of the target system, but can know the resonant frequency by investigating the sensors used by a similar system beforehand.

We illustrate our example in three steps: first, we show the actual crash of the quadrotor under the example attack with a video; second, we explain the low-level data flow compromised by the attack using a code snippet; and lastly, we demonstrate how our framework effectively recovers the quadrotor under the attack with a graph of internal state value changes.

Figure 2 shows the video snapshots of the attack consequence. During the stable flight (the first snapshot), the attack, launched from the second snapshot, compromises the gyroscope sensor measurements of the current angular rate, corrupts the attitude, and causes a sudden increase in the attitude angle. Specifically, the attack corrupts the roll rate to 0.8 rad/s, and then the controller incorrectly tries to change the roll rate to -0.8 rad/s, as it "thinks" 0.8 rad/s is the current measurement. Subsequently, in the next snapshots, the quadrotor under the attack turns over and crashes.

```

1 main_loop() {
2
3     // determines vehicle states
4     angles = read_AHRS();
5
6     // generates target values
7     targets = navigation_logic();
8
9     // generates actuation signal
10    inputs = attitude_controller(targets, angles);
11
12    // sends signals to actuators
13    motor.update(inputs);
14 }
15 read_AHRS() {
16
17    // read IMU sensor measurements
18    for(i=0; i<num_gyro; i++) {
19        gyros[i] = gyro_sensors[i].read(); // *attack*
20
21        // *inserted code for attack recovery*
22        if(abs(soft_gyro[i] - gyros[i]) > k)
23            gyros[i] = soft_gyro[i];
24
25        // weighted sum
26        gyro += w[i] * gyros[i];
27    }
28    // return angles
29    angles = convert2angle(gyro);
30    return angles;
31 }

```

Figure 3: Control loop and sensor reading monitor

Next, Figure 3 shows the simplified code snippet in the quadrotor's control program. The function `main_loop` shows the main control loop, which has a typical feedback control loop structure [21]. Especially, `read_AHRS()` shows a fusion process of gyro sensor readings. It acquires the readings of the multiple gyro sensors via the sensor hardware interface at line 19, and consolidates the information by weighted average at line 26 (various algorithms may use different weights). The data is then processed to obtain the angles (i.e., internal state values) which are returned at line 30.

The sensor attack on the gyroscope compromises the angular rate measurements at line 19. Since `attitude_controller()` generates motor inputs based on the angles from `read_AHRS()`, any compromised gyroscope reading would disrupt the entire control loop. For example, in a stable hover-

ing operation, errors between the current and target angles are minimal. However, the attack compromises the current angles causing an instant increase in the errors. The controller then generates motor input to reduce these fake errors, and consequently introduces unwanted maneuvers. Since the compromised sensor cannot provide the actual valid measurements, the error accumulates over loop iterations.

Our Recovery Approach. Motivated by the sample attack, we propose a *software sensor* based defense technique. We first construct a *system model* that models the behaviors of the controller, actuators, vehicle physics and dynamics. Specifically, it predicts the next physical state given the system input (i.e., reference) and the current state. Software sensors do not interact with the physical environment such that they are immune to physical attacks. Instead, they “measure” the states produced by the system model. In the closed feedback loop as shown in Figure 4, the software sensors are used as standbys: they work in synchrony with the real sensors, and are prepared to take over any time. The recovery switch determines an attack by monitoring the difference between the real and the software sensors measurements, and replaces the real sensors with the corresponding software sensors in the event of an attack. Additionally, if an attack is transient, the switch determines when the attack ends by continuously monitoring the difference, and switches back to the real sensors. Our design is particularly suitable for handling diverse attack scenarios, e.g., attacking one sensor, two sensors (of the same kind), all sensors (of the same kind), and multiple sensors of different kinds, because it detects the ones that misbehave and replaces them with the software version. In contrast, traditional sensor fusion based fault tolerance techniques [4, 16] (e.g., Extended Kalman Filter (EKF) [27]) rely on real physical sensors, including the compromised ones. Therefore, they have difficulties dealing with attacks that corrupt majority sensors of the same kind.

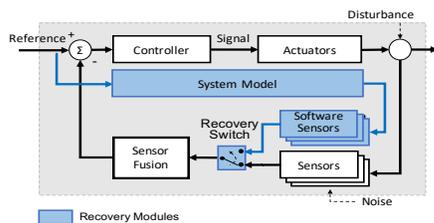
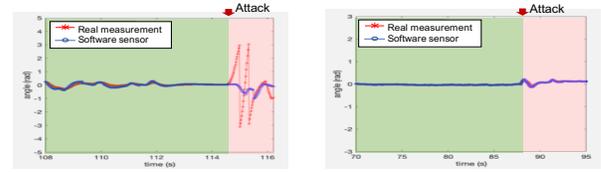


Figure 4: Feedback control loop with our recovery modules

To describe how the recovery modules work, we inserted the attack recovery code at lines 22-23 in Figure 3. The code is placed right after the real sensors readings. At runtime, the code checks if the difference exceeds a pre-defined threshold k , and if so, uses the software sensor measurements instead. The details of software sensor generation and how we distinguish attacks from non-deterministic environmental condition changes will be described in Section 3.2 and Section 4.2.2.

With our recovery modules, the quadrotor can be recovered



(a) Attack without recovery (b) Attack with recovery

Figure 5: Roll changes under the attack

from the attack. Figure 5 shows the changes of roll angle (one of the attitude angles) under the attack with and without the recovery modules during the same mission. The red (star marker) and blue (circle marker) line show the real and software sensor reading, respectively. Before the attack is launched (green area), both are almost identical. However, after the attack, the roll angle is dramatically increased without any recovery action in Figure 5a, whereas with the recovery module in Figure 5b, the software sensor masks and replaces the compromised real sensor measurement. Thus, the recovery modules enable the quadrotor to maintain stable attitude.

Technical Challenges. We should address several prominent technical challenges to use the approach in practice on multi-sensor vehicles: we need to (1) efficiently generate multiple sensor predictions to recover from multi-sensor attacks; (2) consider intrinsic errors such as model inaccuracy and external disturbances (wind, noises, etc.); (3) isolate the specific sensor under attack in time not to propagate corrupted measurements to the vehicle’s internal; (4) set proper recovery parameters such as the recovery switch threshold.

3 Design

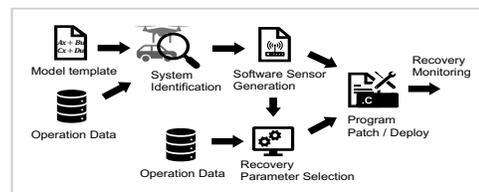


Figure 6: Overview of our recovery framework

Figure 6 presents a high level work-flow of our proposed recovery framework. Each kind of RV, such as quadrotor, hexrotor, and rover, has the same system model template, for example, a polynomial with a specific order and unknown coefficients. The system models of different vehicles (of the same kind) can be considered as various concrete instantiations of the template, that is, polynomials with concrete coefficients. Hence, as the first step of our technique (Section 3.1), given a model template and operation data (i.e., state logs) for a target RV, we leverage system identification [32], a widely used technique to derive system models for the RV. Intuitively, one can consider that it is a training procedure to derive the unknown coefficients such that the model behaviors have minimal errors with the operation log. These

coefficients are jointly determined by the RV’s physical attributes (e.g. weight and shapes), its control algorithm, and the laws of physics. Once the system model is derived, the framework constructs software sensors (Section 3.2) that operate on model responses. Mathematically, these software sensors are also polynomials that take the physical states predicted by the model as input and produce the corresponding sensor readings. For example, the framework employs the model’s angular velocity states to predict the gyroscope sensor’s measurements, thus creating a software-based gyroscope sensor; the reading of air pressure sensor is derived from the altitude prediction of the system model. Software sensor is an approximation and has inherent errors (Section 3.3). Such errors accumulate over time (*drifting*). Hence, we synchronize the predicted states with the real states periodically. Also, our recovery switch utilizes the historical (i.e., accumulated) errors to prevent false alarms and to limit the impact of stealthy attacks by using a small time window. In the next step, we determine the appropriate time window size. The window size is RV specific and hence requires analysis. In addition, we determine the threshold for the recovery switch, which is RV specific and sensor specific. Finally, the framework patches the original control program by inserting the recovery code (Section 3.4) right after sensor reading acquisition.

3.1 System Model Generation

Operation Data Pre-processing. To generate a system model, i.e., a mathematical model reflecting RV’s behavior, we first collect a large corpus of input and output data of the target RV under normal operations; where inputs are the target states, and outputs are the perceived states for the given inputs over time. For the derivation of accurate model, we collect and pre-process the data as follows: First, the data is collected under different maneuvers to appropriately capture various control properties and dynamics. Our mission generator produces random missions systematically based on Mavlink [35] commands. However, since we constrain the model with a template known a priori, the amount of data needed by our approach is much less than an ML-based learning approach [26, 46] – only those involved in the template are needed. Second, the data is collected at a high sampling rate to adequately reflect the highly reactive behavior of the RV to the surrounding physical environment. However, as the log system uses substantial system resources for saving values to memory (e.g., flash card or disk), the typical log update rates are lower than the control loop frequency with a limited number of variables. Specifically, various RV components have different update frequencies - e.g., 400Hz sampling rate for critical sensors, 100Hz sampling rate for non-critical sensors and RC modules, and 10Hz update rate for the RV’s own log module. As such, aligning these different data streams is a prominent challenge. To address the problem, we convert various data streams to the same target frequency us-

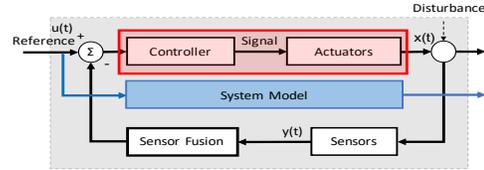


Figure 7: System model in closed loop

ing a resampling technique. It interpolates new data points within the range of existing sample points by minimizing overall curvature, resulting in a smooth line that passes the existing samples. Here, we use spline interpolation, to avoid Runge’s phenomenon [3] which causes oscillation in high degree polynomials. Offline resampling enables us to obtain high frequency data without additional runtime overhead.

Model Construction. The template of an RV’s system model consists of the state and output equations, i.e., Eq. (1) and Eq. (2), respectively:

$$\dot{x}' = Ax(t) + Bu(t) \quad (1)$$

$$y(t) = Cx(t) + Du(t) \quad (2)$$

where $u(t)$ is system input (i.e., the target state as shown in Figure 7), and $y(t)$ is system output. Output $y(t)$ is measured by sensors. The model specifies how the physical states $x(t)$ of the system respond to external inputs and control signals with the underlying control algorithm and system dynamics. As shown in Figure 7, the system model (in the blue box) can be considered a counterpart of the combination of control algorithm, actuators, and vehicle dynamics (in the red box). We leverage the system identification (SI) technique [32] for deriving the system model, which is widely used in different applications [5, 56]. Given the model template (Eq. (1), (2)) and a large set of collected operation data, SI instantiates the A, B, C, D matrices so that the resulting equations produce the best fit for the data. We use the SI Toolbox by MATLAB [34]. Note that the system model is not a software sensor. The output of the model should be accordingly converted to the individual sensors with our on-the-fly operation and error correction technique.

Example. For a quadrotor system, we can generate the models for individual state variables, defined by the following:

$$x = [x \ y \ z \ \phi \ \theta \ \psi \ \dot{x} \ \dot{y} \ \dot{z} \ p \ q \ r] \quad (3)$$

where $[x \ y \ z]$ is the position vector, $[\phi \ \theta \ \psi]$ is the attitude angles (roll, pitch, yaw), $[\dot{x} \ \dot{y} \ \dot{z}]$ is the vehicle velocity, and $[p \ q \ r]$ is the vehicle angular velocity. For each variable, we first determine the state and output template equations. Specifically, we use a discrete-time state-space model template, encoding a PID controller and dynamics equations known a priori for the family of the subject RV. Then, for each variable, we specify a model order (i.e., the degree of polynomial equations). We then employ SI to instantiate the unknown coefficients of the template using an iterative prediction error minimization

algorithm [32]. SI in our technique is not limited to the linear state-space modeling. A non-linear model can also be derived from the known template. However, for our purpose (e.g., rigid body RVs), the linear-model is sufficient to approximate the actual higher-order close-loop dynamic, since the dominating system dynamic is a second-order system. Even for an advanced non-linear control algorithm, the control effort is mainly from the linear portion, namely proportion, derivative, and integral [24, 55].

We note that our model construction is generic, since the same model template can be used to instantiate the models for a family of vehicles with a similar physical structure. Besides, our methodology is efficient. Given the profile data and known model template, SI can optimize the coefficients and derive a state-space model that accurately predicts the next states with reasonable computation time (Section 4.2). Our model construction is different from most SI applications in control systems, which often focus on modeling the vehicle dynamics, whereas ours models both dynamics and control algorithm.

3.2 Software Sensor Construction

Software sensor is a software-based virtual sensor which generates the prediction of the corresponding real sensor measurement. It predicts real sensor reading based on the system model output, i.e., the predicted new physical state. Since the physical state prediction is completely model-based, software sensors are independent of real sensor measurements that are vulnerable to physical attacks. Specifically, at run-time, software sensor readings are compared with real sensor measurements. Once an attack is detected on some physical sensor (i.e., its real measurement differs significantly from the predicted one), the corresponding software sensor is used to replace the real one. An RV often has many kinds of physical sensors. Their software version may require non-trivial derivation from the system model outputs. In the following, we explain the mathematical conversions entailed by software sensors.

Conversion Operation. We provide the conversion operations for various sensors (accelerometer, gyroscope, barometer, magnetometer, GPS).

An accelerometer measures linear acceleration of the vehicle. However, the outputs of our example model contain only 12 states that do not directly include acceleration information. Therefore, a conversion operation (Eq. (4)) is required.

$$a(t) = c_k \frac{v(t) - v(t-k)}{k \cdot \Delta t} \quad (4)$$

where v is the velocity, Δt is the sampling time interval (temporal distance between two samples), c_k is a constant coefficient and k is the number of equidistant sample points; k is usually much larger than 1 to tolerate the noise induced by high frequency sampling.

A gyroscope measures angular velocities which are critical in maintaining stable movement, especially for aerial vehicles. To obtain accurate measurements, the gyroscope in IMU operates with a high sampling rate. Other sensors further help to correct gyroscope sensor errors to estimate accurate orientation state (i.e., attitude angles). Gyroscope intrinsically has *drift error* over time due to an integration operation over angular velocities for obtaining angles. In the recoverability test of our approach under the different combinations of attacks on multiple sensors (see Section 4.2.2), gyroscope sensor is the most sensitive and requires accurate prediction for recovery. In this case, it turns out that using software gyroscope alone is not sufficient (leading to reduced stability and operation time) when all the physical gyros are compromised. As such, we introduce a compensation approach to improve accuracy by leveraging other types of real sensors. The details are shown in Section 3.3.

A barometer measures atmospheric pressure, which is necessary to determine altitude. We use Eq. (5) to calculate air pressure from altitude (position z in the system model states).

$$P_h = P_0 \cdot \exp \left[\frac{-g_0 \cdot M \cdot (z - h_0)}{R \cdot T_0} \right] \quad (5)$$

where P_0 is the base air pressure (Pa), g_0 is the gravitational acceleration ($9.87m/s^2$), M is the molar mass of Earth's air ($0.02896kg/mol$), h_0 is the base altitude, R is the universal gas constant ($8.3143N \cdot m/mol \cdot K$), T_0 is the base temperature (K), and z is the current altitude from the model states.

A magnetometer, also known as compass, measures the strength of the Earth's magnetic fields in 3-axis, which is used to calculate orientation (heading) information. The following equation shows the transformation of the magnetic fields to orientation status (i.e., the heading direction of the RV):

$$H = \text{atan2}(-m_y \cdot \cos\phi + m_z \cdot \sin\phi, m_x \cdot \cos\theta + m_y \cdot \sin\theta \cdot \sin\phi + m_z \cdot \sin\theta \cdot \cos\phi) \quad (6)$$

where H is the heading direction yaw, and m_x, m_y, m_z are the magnetic field measurements along each axis. Control systems do not directly use the magnetic field measurements, but rather rely on the extracted orientation. Therefore, instead of converting the system model responses to raw magnetic field sensor measurements, we directly use the orientation states from the system model.

Global Position System (GPS) measures geometric positions and velocities which collectively enhance the position and attitude estimation along with other sensors. GPS measurements can be directly acquired from the system model.

Coordinate System Transformation. Based on the system model responses and conversion equations, we can approximate sensor measurements. Note that internal state variables and sensor measurements may be aligned with different reference frames. Intuitively, each frame can be considered a different coordinate system. Information can be exchanged/aggregated only after they are projected to the same coordinate

system [33]. Hence during software sensor conversion, we have to perform frame canonicalization. Specifically, for an RV with a rigid body, we commonly use different reference frames for describing its position and orientation (i.e., pose). The inertial and body frames are used to provide the pose in the global and local coordinate systems, respectively. The inertial frame is an earth-fixed frame, whereas the body frame is aligned with the vehicle’s body (hence the sensors). The sensor measurements are usually related to the body frame where the sensors are attached to, and must be converted from the body frame to the inertial frame and vice versa. Frame conversion is accomplished by multiplying with constant conversion matrices. The detailed equations are in Appendix A.

3.3 Error Correction

Software sensors aim to closely predict real sensor measurements. However, the errors between software and real sensors are intrinsic for the following reasons: (1) the conversion (from model states to sensor readings) introduces *conversion errors*, (2) the system model provides only an approximation of the real states and hence introduces *model errors* over time, (3) external disturbances and noises affect the accuracy of model prediction, which introduce *external errors*.

Obtaining an accurate prediction model - thus avoiding the above errors - through precise modeling of complex real-world effects for a specific system is neither practical nor generic. Instead, we choose to tolerate model inaccuracies through integrating additional error correction techniques to compensate for the errors. Note that our recovery does not aim to replace the real sensors permanently when the attack is continuous, but rather aims to isolate the compromised sensor and provide the needed feedback to the control loop for a certain time duration so that we can ensure continuous stable operation for some time without catastrophic consequences (e.g., immediate crashes) or take an appropriate emergency action (e.g. safe landing).

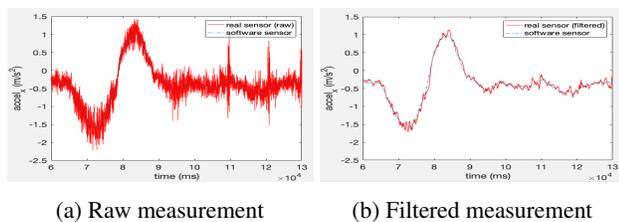


Figure 8: Raw and filtered acceleration measurements with software sensor output

Conversion Errors. Although the concept of comparing software sensor readings and real readings is straightforward, direct comparison is problematic. Specifically, on the software sensor side, higher-order state variables (e.g. acceleration) may contain noise at high-frequency. The conversion process introduces additional errors. As such, directly comparing such software sensor readings with the real ones leads to numer-

ous false alarms. To address the problem, we leverage error reduction techniques [40]. Specifically, to mitigate output inaccuracy caused by numerical differentiation, we can use a simple finite differentiation method like Eq. (4). However, with this method, the output tends to be close to zero in the presence of high frequency noise. To tackle this, we implement a smooth noise-robust differentiator that provides noise suppression [25].

On the real sensor side, raw measurements have various kinds of errors, such as noise, bias, and time lags. Figure 8a shows that a highly fluctuating raw signal is not ideal for comparison with the software sensor signal. Therefore, we smooth it out with a basic filter (see Figure 8b). Specifically, we apply the low-pass filter [40] which is a standard filter to attenuate high frequencies with a pre-selected cutoff.

Model Errors. The system model approximates the real RV states. Such approximation contains intrinsic *model error*. This is because the model is constructed from a universal template (for a family of vehicles), which does not describe the details and nuances of a concrete RV. In addition, the model assumes a simple linear PID controller whereas real RVs may use non-linear control algorithms. To mitigate intrinsic model errors, we introduce periodic synchronization and error reset. Although model errors are marginal at any time instance, they tend to aggregate overtime, namely *prediction drift*. Thus, errors should be corrected periodically.

Specifically, our solution regularly resets software sensor errors, by synchronizing with the real sensor readings to remove prediction drift during normal operations. To reset errors, we partition the entire operation duration to small time windows of a fixed duration and synchronize the software sensor readings with the real ones at the start of each window. Note that the synchronized readings are then fed to the system model, eliminating errors in the predicted system states.

Recovery Parameters. We select the recovery parameters (i.e., window sizes and recovery thresholds) systematically. The window size (N) for historical error is an important parameter. If N is too large, there can be a significant accumulation of the error which could cause false alarms. Conversely, if N is too small, the synchronization of the software sensors with the real sensors will be so frequent that it would lead to false negatives. Moreover, the conversion might introduce a small delay in the generation of software sensor measurement, causing it to not align with the real sensor. Therefore, to achieve the measurement synchronization at correct time-steps, N should be more than a potential time-displacement between the software and real sensor signals. We choose N to be the maximum time-displacement computed from the large set of operation data using the dynamic time-warping algorithm [45] that computes the optimal alignment of two data sequences. Once the window size is determined, we calculate the maximum error between two signals within each window in the large set of operation data. We select the threshold $T = e_{max} + m$, where e_{max} is the maximum accumulated error

and m is a margin parameter. For example in the 3DR Solo quadrotor, the main control loop is invoked at every 2.5ms and we use the 575ms (i.e., 230 loop counts) as the window size which is chosen as described above. We evaluate the effect of different recovery parameters in Section 4.2.

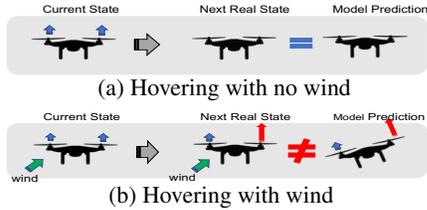


Figure 9: External force and *external error* (state discrepancy)

External Errors. So far, our technique does not model external forces that may introduce errors. For example, when the wind speed is 5 mph west, it introduces external forces that move the aerial vehicle to the east. To adapt software sensors to external disturbances, we calculate an estimation of external force by measuring errors within the previous window. The key observation is that external forces are likely static across two consecutive windows in practice as long as the wind is not drastically changing. Specifically, the time unit is 400Hz (2.5ms), and the window is 2Hz (500ms) less. Within 0.5s, the forces are highly likely unchanged.

Based on our observation, we calculate the average error by comparing the real state and model prediction within each window. Figure 9 illustrates a simple external error. The quadrotor maintains a stable attitude during hovering. Without wind (a), the real next states and model prediction are the same since both cases are affected only by thrust force (the blue up-arrow). However, with the wind (b), to maintain a stable state without tilting, the controller increases the right thrust (in red) such that the drone does not tilt right, whereas the model (without the wind force) thinks the vehicle tilts left with the increased right thrust. We use the average error from the previous window as an estimate of external forces in the next window. The correction result can be found in Section 4.2.2.

Supplementary Compensation. Certain sensor types and usage scenarios require very accurate measurements. As such, using software sensors alone may not be sufficient, especially for lengthy operations. To increase an accuracy of these special sensors and to extend the operation time under the recovery mode, we employ additional error correction techniques. Specifically, we leverage other types of sensors - less sensitive ones - to reduce the estimation errors. Under this approach, we can provide real sensor readings in model prediction. Since model prediction is based on the model and real input, the real sensor measurement (converted from different sensors) would contain more realistic feedback with real disturbances factored in. We present an example of this approach that estimates angle status from the accelerometer and magnetometer (where angle status is typically measured by gyro) in Appendix B. Note that this compensation approach is not

Algorithm 1 Runtime Recovery Monitoring

```

1:  $u$  control input of the real vehicle
2:  $m$  sensor measurement
3:  $x$  control states of the real vehicle
4:
5: procedure RECOVERYMONITOR( $u, m$ )
6:    $y \leftarrow C \cdot x + D \cdot u$  ▷ calculates model response
7:    $x \leftarrow A \cdot x + B \cdot u$ 
8:    $m \leftarrow \text{filter}(m)$ 
9:    $m_s \leftarrow \text{convert}(y)$ 
10:   $t++$ 
11:  if  $\text{!recovery\_mode} \ \&\& \ t > \text{window}$  then ▷ checkpoint
12:     $t \leftarrow 0$ 
13:     $r \leftarrow 0$ 
14:     $e \leftarrow \text{error\_estimation}(r, m, m_s)$ 
15:     $m_s = m$ 
16:  end if
17:   $m_s \leftarrow m_s - e$  ▷ error compensation
18:   $r \leftarrow r + |m - m_s|$ 
19:  if  $r > T_{on}$  then ▷ checks residual
20:     $\text{recovery\_mode} \leftarrow \text{true}$ 
21:     $\text{safe\_count} \leftarrow 0$ 
22:  end if
23:  if  $\text{recovery\_mode}$  then
24:     $m \leftarrow m_s$  ▷ recovers sensor
25:    if  $r < T_{off}$  then
26:       $\text{safe\_count}++$ 
27:    end if
28:    if  $\text{safe\_count} > K$  then ▷ switches back
29:       $\text{recovery\_mode} \leftarrow \text{false}$ 
30:    end if
31:     $\text{recovery\_action}()$  ▷ optional action
32:  end if
33: end procedure

```

necessary for majority of the sensors. In most cases, using our software sensors for recovery - without other real sensors - is sufficient. In our evaluation (Section 4.2.2), among all the studied scenarios, we leverage this technique only when all the gyros are compromised at the same time, which is rare.

3.4 Recovery Monitoring

Algorithm 1 describes our proposed recovery procedure. The `recovery_monitor()` function is inserted right after the sensor reading code in the main control loop. It takes runtime inputs and actual sensor measurements as the parameters. It then computes the predicted new state (x) and output (y) from the previously predicted states (line 6 and 7). The real sensor measurements (m) are first filtered to attenuate noises (line 8). The model output is then converted into sensor prediction (m_s) according to the sensor type (line 9). In lines 11-16, when the current time is a checkpoint, that is, the start of a new window, the error (e) is calculated using the function `error_estimation()` that estimates the model and external error (see Section 3.3). Error compensation is applied to the sensor prediction within the window (line 17). At line 18, the cumulative difference (residual r) is computed by comparing the values with the real measurement. If the difference exceeds the recovery threshold T_{on} , then it changes to recovery mode and starts new `safe_count` (line 19-22). In the recovery mode, the real sensor measurement (m) is replaced by the sensor prediction (m_s) (line 24). At the same time, the difference is continuously checked by the recovery-off threshold T_{off} (usually, $T_{off} < T_{on}$) and when the difference is smaller

than the threshold, the *safe_count* is increased (lines 25-27). When the difference is below the threshold for more than K times, we resume using the real sensors, assuming the attack is over (lines 28-30). If there is predefined recovery action, we trigger it through calling the function *recovery_action*.

4 Evaluation

We have developed a prototype that includes a mission generator based on Mavlink [35], a customized log module using Dataflash log system, and a system model construction component implemented in Matlab. The recovery module is implemented in C/C++ and includes the software sensors, recovery switch, error correction modules (i.e., differentiator, low-pass filter and supplementary compensation). The model validation and parameter selection components using the profile data are implemented in Matlab. Additionally, we implemented attack modules to simulate physical sensor attacks that maliciously modify the sensor measurements via a remote trigger at runtime.

4.1 Evaluation Setting

We evaluate our framework with both simulated and real-world RVs, including quadrotor, hexarotor, and rover. Table 1 shows the subject vehicles. We first evaluate the effectiveness of our technique under various simulated environmental conditions, since it is difficult to realize different wind effects/conditions in real-world. We then confirm the results with real vehicles including a 3DR Solo quadrotor and an Erle-Rover in real-world conditions.

Table 1: Subject Vehicles in Evaluation

Type	Model	Controller Software	Number of Sensors				
			G	A	M	B	P
Quadrotor	APM SITL	ArduCopter 3.4	2	2	1	1	1
Hexacopter	APM SITL	ArduCopter 3.6	2	2	1	1	1
Rover	APM SITL	APMrover2 2.5	2	2	1	1	1
Quadrotor	Erle-Copter	ArduCopter 3.4	2	2	1	1	1
Rover	Erle-Rover [†]	APMrover2 3.2	1	1	2	1	1
Quadrotor	3DR Solo [†]	APM:solo 1.3.1	3	3	3	2	1

^{*} G: gyroscope, A: accelerometer, M: magnetometer, B: barometer, P: GPS

[†] Real Vehicles

Real Testbed. Our real testbed consists of two commodity RVs: a 3DR Solo [1] and an Erle-Rover [18]. 3DR Solo is a typical commercial quadrotor that leverages heterogeneous and redundant sensors for flight stability. The aerial vehicle is highly dynamic and can be easily affected by environmental factors. The 3DR Solo system is implemented in Pixhawk 2 from the open-source autopilot project Pixhawk [44], and uses APM:Copter, an open-source flight controller based on the MAVlink protocol and part of the ArduPilot project [2]. Erle-Rover is equipped with various sensors and is a representative ground RV. Erle-Rover is implemented with Erle-Brain 3, a linux-based system provided by Erle Robotics. We use the open-source control software APMrover 2 for the rover. Table 2 lists the sensors in 3DR Solo and Erle-Rover. 3DR

has 12 sensors and the rover has 6. Note that many sensors are replicated with different hardware to avoid the same type of failures. For example, 3DR has three gyroscope sensors manufactured by different vendors. To compromise the entire set of sensors of the same type, the attacker should have different attack techniques.

Attack and Recovery Setting. To generate the physical sensor attacks discussed earlier (See Section 1), we insert attack modules into the firmware. Since it is difficult to implement the actual hardware attacks which require special devices, we simulate the same effects with the attack modules - but the actual attack does not access internals. Specifically, we add a piece of malicious code into the sensor interface that transmits the sensor measurements to the main closed control loop. The attacks modify sensor measurements (through attack code) to mimic the effect of real "controlled attacks" that control sensor readings (e.g, a sinusoidal wave, random or selected values). Moreover, we consider continuous attacks rather than instantaneous attacks since temporary attacks can be easily recovered by our method. We map Mavlink commands to various attack types to remotely trigger via the ground control.

We say recovery is successful, when after an attack is launched, the technique detects it and triggers the recovery logic to ensure the current states are within a certain error bound of the expected states for a certain period of time:

$$R_{succ} := |Y_t - \bar{Y}_t| \leq \epsilon, t \in [1...k] \quad (7)$$

where Y_t is the real output, \bar{Y}_t is prediction, ϵ is the error margin, t is the timestamp in the recovery mode, and k is the maximum time to decide recovery success. For example, $\epsilon = 3$ and $k = 10$ indicate that a RV performs missions within 3 meters error for 10 seconds under the recovery mode.

Note that our recovery technique does not consider the previous maneuvers (at $t \leq 0$) since our software sensors accurately predict the real measurements in the "various maneuvers" (Figure 12). As long as recovery starts with the accurate initial states via software sensors, subsequent sensor feedbacks are precise and the control loop can obviously control the vehicle to stable states and recover. Also, our goal is to prevent immediate crash and provide the transition time for emergency operation (e.g., manual mode), not to replace the compromised sensor permanently. Therefore, after recovery mode on, the vehicle would conduct stable operation (e.g., hovering) before changing to the emergency operation.

4.2 Experiments and Results

4.2.1 Efficiency

In terms of the space overhead, we measure the firmware size before and after our recovery code is inserted. For the runtime overhead, we compare the execution time of the main control loop before and after. Specifically, we first measure the (space and runtime) cost of the original code as a baseline, which

Table 2: Sensors in 3DR Solo quadrotor and Erle-Rover

Vehicle	Sensors	Manufacturer	Model	Location	Measurement	Data Type	Frequency
3DR Solo	Gyroscope1	InvenSense	MPU6000	Pixhawk 2 (onboard)	Angular Rate	Angular Motion	400Hz
	Gyroscope2	InvenSense	MPU6000	Pixhawk 2	Angular Rate	Angular Motion	400Hz
	Gyroscope3	STMicroelectronics	L3GD20	Pixhawk 2	Angular Rate	Angular Motion	400Hz
	Accelerometer1	Measurement Specialties	MPU6000	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Accelerometer2	InvenSense	MPU6000	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Accelerometer3	STMicroelectronics	LSM303D	Pixhawk 2	Acceleration	Linear Motion	400Hz
	Magnetometer1	Honeywell	HMC 5983	Pixhawk 2	Magnetic Field	Angular Position	100Hz
	Magnetometer2	STMicroelectronics	LSM303D	Pixhawk 2	Magnetic Field	Angular Position	100Hz
	Magnetometer3	Honeywell	HMC 5983	Body (Leg)	Magnetic Field	Angular Position	100Hz
	Barometer1	Measurement Specialties	MS5611	Pixhawk 2	Air Pressure	Linear Position	50Hz
	Barometer2	Measurement Specialties	MS5611	Pixhawk 2	Air Pressure	Linear Position	50Hz
	GPS	u-blox	NEO-7M	Body (Head)	Position	Linear Position	50Hz
Erle-Rover	Gyroscope	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Angular Rate	Angular Motion	50 Hz
	Gravity Sensor	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Acceleration	Linear Motion	50 Hz
	Compass 1	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Magnetic Field	Angular Position	10 Hz
	Compass 2	u-blox	Neo-M8N	External (roof)	Magnetic Field	Angular Position	10 Hz
	Pressure Sensor	Erle Robotics	Erle-Brain	Erle-Brain 3 (onboard)	Air Pressure	Linear Position	10 Hz
	GPS	u-blox	Neo-M8N	External (roof)	Position	Linear Position	50 Hz

does not include the recovery code. Then, for each sensor, we insert the recovery code including the required libraries that correspond to the sensor (e.g., filters and utility functions) and measure the overhead. Finally, we insert all the recovery code for all sensors to obtain the total overhead (for all the simulated and real RVs).

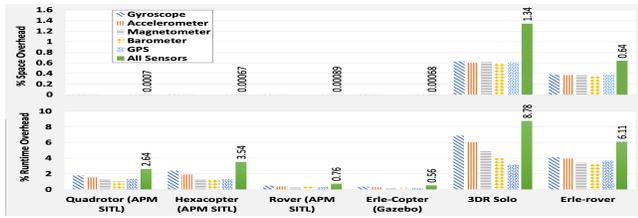


Figure 10: Space and Runtime overhead

Space Overhead. Unlike traditional computing systems, RVs usually have limited memory space. As such, code size is an important performance factor. As shown in Figure 10, the increase of code size (i.e., additional firmware size needed) incurred by our recovery modules is marginal. The space overhead is at most 1.3% when all software sensors are loaded and less than 0.7% for individual sensors. Note that some code pieces are shared across software sensors. The simulated vehicles have negligible overhead since the executables are relatively larger than those of the real vehicles.

Runtime Overhead. We measure the average per-iteration execution time of the main loop which includes various control functions and auxiliary tasks. In ArduCopter and APM-rover2, the system loop execution frequency is 400Hz and 50Hz respectively. Every 2.5ms or 20ms, the scheduler executes the control functions, and then schedules auxiliary tasks using the remaining time in the epoch. Basically, all the tasks should be completed within the hard deadline (i.e., 2.5ms or 20ms). Figure 10 shows the results. The runtime overhead introduced by the recovery module for single sensor recovery is at most 6.9%, whereas, for multiple sensor recovery, the total overhead is at most 8.8%. We also consider the CPU utilization rate (for real vehicles), which is the iteration execution time over the hard deadline. For the 3DR Solo, the rate

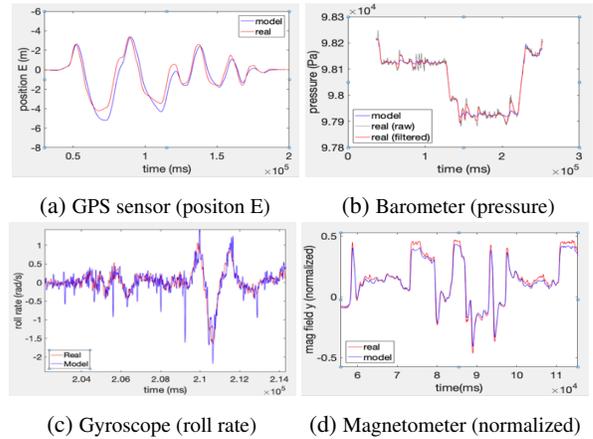


Figure 11: Sensor prediction

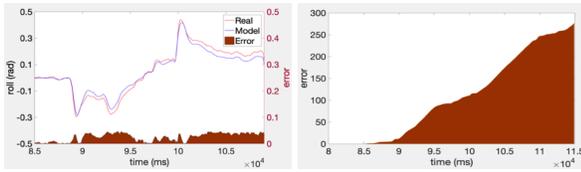
increases from 63.32% to at most 67.68% (i.e., by 4.36%) for single sensor recovery, and to 68.88% (i.e., by 5.56%) for multiple sensor recovery. For the Erle-rover, the rate increases from 26.7% to at most 27.8% (i.e., by 0.9%), and to 28.4% (i.e., by 1.7%) for single and multiple sensor recovery, respectively. Note that the observed overhead does not impact normal operations, as the per-iteration runtime does not exceed the hard deadline. Real recovery cases in Section 4.3 demonstrate that our technique is practically effective.

4.2.2 Effectiveness

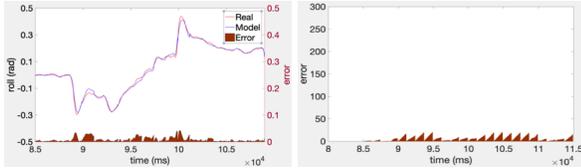
We evaluate effectiveness as follows. (1) We first show software sensors can precisely predict real sensor measurements under various maneuvers; (2) we show that the error correction techniques can effectively attenuate the prediction errors; (3) we demonstrate that parameter selection is effective; (4) we show that our framework can successfully recover from multiple attacks with real vehicles in real environments; (5) last, we further evaluate our technique under various environmental conditions and attack scales.

Software Sensors. Figure 11 shows how closely software sensors predict (blue lines) the real readings (red lines) in the various maneuvers of 3DR Solo. The figures for other

systems are similar and hence omitted. It can be observed that there are errors (e.g. drift and external error) between the predictions and the real measurements, which we will remove using the error correction techniques demonstrated below.



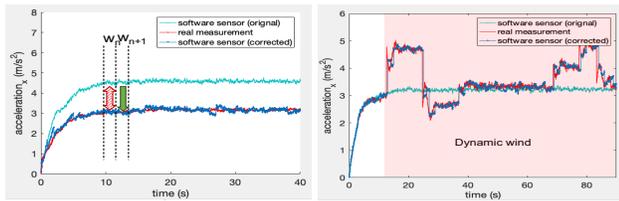
(a) Roll prediction and accumulated errors without correction



(b) Roll prediction and error correction with synchronization

Figure 12: Drift correction with synchronization and error reset

Error Corrections. Figure 12 shows the drift in the roll angle prediction before and after error correction. We measure the roll value and the prediction error during a real flight of the quadrotor (left of (a)). As shown in (b), at each window start, the initial state is synchronized, and the accumulated error is reset. As such, the accumulated error is significantly reduced (right of (b)). In this experiment, we used 1.0s window size with the main sampling rate $T_s = 2.5ms$. The results for other sensors and vehicles are similar.



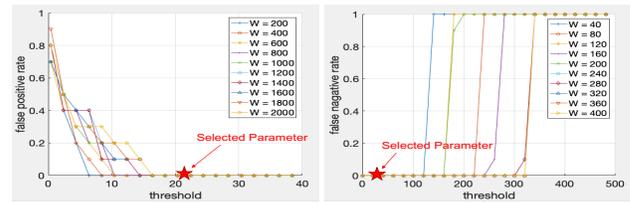
(a) Constant wind

(b) Dynamic wind

Figure 13: External force (wind) corrections for different winds

Figure 13 shows the external force estimation and correction for when there is wind. Here, a simulated APM quadrotor is flying north. We use a simulated RV as we need to create different windy conditions. First, we generate an artificial wind towards the south with a constant velocity 29mph (i.e., a strong breeze in Beaufort scale 6). As the wind pushes the vehicle to the opposite direction, the real acceleration measurement is lower than the software sensor which does not model the wind. We correct this external error in software sensor by subtracting the average error in a window from the following window predictions (see Figure 13a). Similarly, we present the error correction under a dynamic wind - composed of randomly generated wind portions with a random selection of speed ([10...30]mph), direction (N,S,E,W) and the portion duration ([2..10]s) - in Figure 13b. Observe that software

sensors can produce accurate predictions after correction.



(a) False positives rates

(b) False negative rates

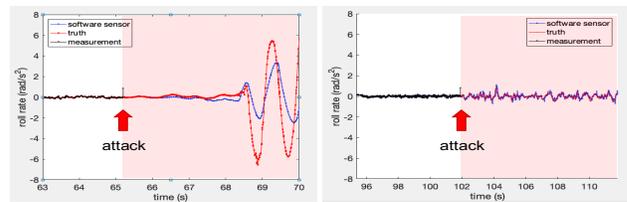
Figure 14: Different recovery parameters and FP/FN rates

Parameter Selection We study the effect of recovery parameters (i.e., window size and recovery switch threshold) on the recovery mode activation. We generate 20 missions (i.e., a sequence of primitive moves like straight fly, turns, etc.) with no attack, to measure the FP rates (i.e. how many times recovery is activated), and 20 missions with the injected attacks to measure the FN rates (i.e. how many times recovery activation is missed). Figure 14 shows the results for different parameter values. Observe that, (1) for a given window, the larger threshold raises less FPs and more FNs; (2) for a given threshold, the larger window causes more FPs and less FNs. Also note that the values chosen by our parameter selection strategy (see Section 3.3) lead to zero FPs and FNs.

Table 3: Attack combination and recovery result

Test#	GPS	Barometer 1 2	Gyroscope 1 2 3	Recovered
C1	Compromised	Benign	Benign	✓
C2	Benign	Compromised	Benign	✓
C3	Benign	Benign	Compromised	✓†
C4	Compromised	Compromised	Benign	✓
C5	Compromised	Benign	Compromised	✓†
C6	Compromised	Compromised	Compromised	✓†

✓: success, †Supplementary Compensation Applied



(a) without compensation

(b) with compensation

Figure 15: all-gyroscopes attack recovery and compensation

Multiple Sensor Attacks and Results. We perform combinatorial attacks on heterogeneous sensors of 3DR. For the same type of sensors, we attack the entire set of sensors at the same time. Table 3 shows the results. In this experiment, we observe that the sensors have different sensitivity and importance. When GPS and Barometers are compromised, we can recover the vehicle. However, when all gyros are attacked (C3, C5, and C6), the recovery duration was short (3 sec).

To further investigate the gyroscope recovery case, we perform different attacks on the available gyroscope sensors. Table 4 shows the results and the comparison with a traditional fail-safe mechanism (i.e., TMR). We can recover from

Table 4: Attack on attitude (gyro) sensors and recovery result

Test#	Gyroscope 1	Gyroscope 2	Gyroscope 3	Proposed	TMR
T1	Compromised	Benign	Benign	✓	✓
T2	Compromised	Compromised	Benign	✓	✗
T3	Compromised	Compromised	Compromised	✓ ¹	✗

✓: success, ✗: fail to recover, 1: recovery with supplementary compensation

the attack on a single gyro (Test T1) and the attack on the majority gyros (more than half – Test T2) without supplementary compensation. When all gyro sensors are compromised, we can recover from the attack with the complementary approach, leading to increased recovery duration. In comparison, the traditional fail-safe mechanism fails to recover when the majority of gyro sensors are compromised. In general, some RV systems are equipped with failsafe modes (e.g., emergency landing or manual mode). However, those approaches still rely on the remaining benign sensors, or otherwise they undergo immediate crashes even with the failsafe mode.

To increase the recovery duration, we leverage our compensation approach described in Section 3.3. Specifically, to compensate for the accuracy loss, the gyroscope readings are combined with readings from other types of sensors. Figure 15 shows that the internal state (i.e., roll rate) changes during recovery *without and with* compensation. The red curve represents the actual physical state (ground truth roll rate) of the vehicle in real-world. Under attack, the software sensor (blue curve) replaces the real measurement (black curve). However, without compensation, a small error in the software sensor prediction accumulates over time and causes the actual roll rate to oscillate significantly after a few seconds (see Figure 15a). Note that our recovery technique prevents an immediate crash even without compensation applied. Our compensation approach increases the software sensor accuracy by adding the supplementary measurements, leading to a more stable roll rate and a longer recovery duration (see Figure 15b). The video is available at [12]. More discussion of the recovery cases can be found in Section 4.3.

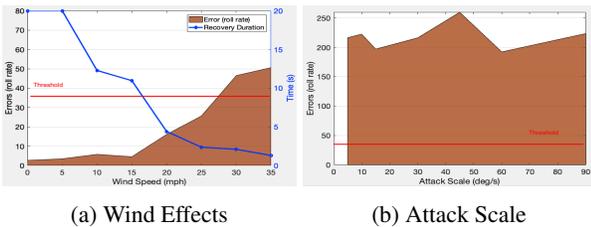


Figure 16: Errors under different wind speeds and attack scale

Wind Effects and Attack Scale We evaluate the error (i.e., the difference between software and real sensors) under different wind speeds and attack scales while the vehicle performs missions including straight flies and turns. We also measure the duration of stable operation after recovery, called the *recovery duration*. Specifically, we artificially inject wind in the simulation with different speeds from 0 to 35mph. For the different attack scales, we change the maliciously injected roll rate from 5 to 90deg/s. To measure the recovery duration, we

find the value of k in Eq. (7) with $\epsilon = 3$ and maximum $t = 20$. Figure 16a shows that the error (brown area) is small with small (0-7mph) and moderate (8-15mph) wind, and significantly increases with strong wind (above 20mph). Observe that the errors are lower than our recovery threshold (=38) with in most case. Only the very strong wind (above 27mph - a gale in Beaufort scale) generates errors exceeding the threshold. However, commodity drones are recommended not to fly in wind speed exceeding 22mph. And, Figure 16b shows the error during the gyroscope attack. Observe that it is significantly larger than the recovery threshold (=38) for all the attack scales. The magnitudes of errors in the two figures demonstrate that the our selected threshold value can distinguish between wind and attack. Lastly, as shown by the blue curve in Figure 16a, the recovery duration is at least 10 seconds with small/moderate wind. When the wind is strong (> 20mph), the duration is significantly reduced.

4.3 Case Studies

We present case studies with the two real RVs with four different attacks under various movements: gyroscope and GPS attacks on the 3DR Solo, two GPS attacks on the Erle-rover.

Gyroscope recovery on 3DR Solo. In this attack, 3DR Solo takes off from home and maintains its position at a predefined altitude (i.e., hovering). Then, we launch an attack on its available gyroscope sensors (3 in total). Specifically, we insert constant values (under the attacker’s control) to disrupt the gyroscope roll rate measurements. Without our recovery framework, the vehicle instantly deviates from its hovering condition and crashes. Figure 17a show the trajectory in red (top) and roll rate changes (bottom) during the mission. The green region indicates the normal operation without the attack while the red region shows the roll rate changes under the attack. When the attack is launched, the controller uses the compromised roll rate ($\sim 0.6rad/s^2$) and accordingly tries to reduce the rate to match the target state (zero). This conversely decreases the roll rate leading to overturning and consequently crashing the vehicle. With recovery, the software sensor prevents the crash by providing the proper feedback to the control loop under the attack. Figure 17b shows the real trajectory (top) and roll rate changes (bottom) during the attack and recovery. During the normal operation, the measurement (blue curve) takes the real sensor readings whereas under the recovery mode (after the attack), the measurement takes the software sensor (red curve). Observe the software sensor introduces a reasonable amount of oscillation that was not seen under the real benign sensors because of inevitable inherent errors. However, despite the errors, it still allows the vehicle to maintain the hovering position under the attack, preventing the crash. Videos are available at [9].

GPS recovery on 3DR Solo. We compromise the GPS sensor reading with a more complex mission. The 3DR Solo performs a waypoint navigation mission where it flies through

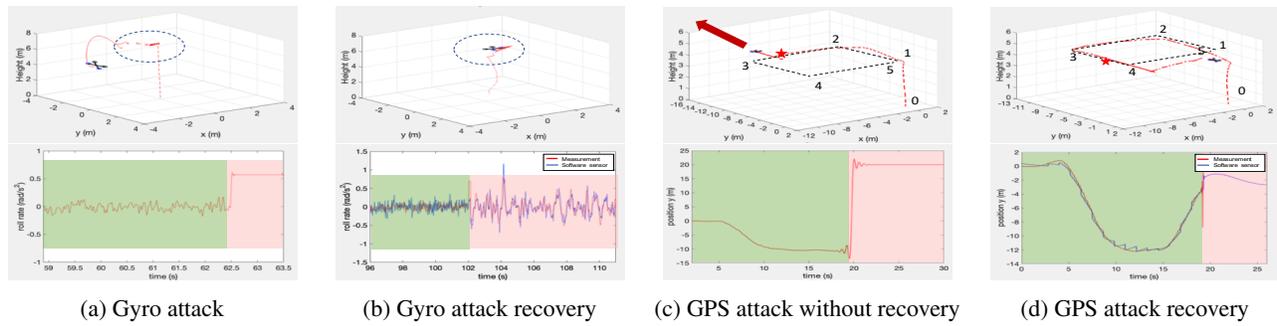


Figure 17: Attack and recovery under the sensor attacks on 3dr Solo

five waypoints in a square shape trajectory after take-off. We launch the attack by modifying the longitude positional information of the GPS measurement - set to 20 meters left from the actual position. As shown in Figure 17c, the vehicle deviates from the expected trajectory (black line) and flies to the right (red star), due to the compromised measurements. With recovery, Figure 17d shows that the vehicle continues its planned mission (with a marginal deviation) as the compromised measurements were replaced by the software sensor. Videos can be found at [10] and [15], respectively.

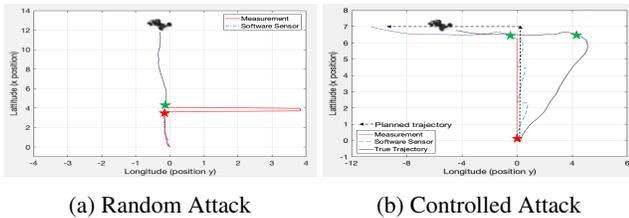


Figure 18: GPS attacks and recovery on Erle-rover

GPS recovery on Erle-rover. We conduct two different GPS attacks on Erle-Rover: *random* and *controlled* GPS attacks.

In the *random* attack, we inject random signals to compromise positional information (i.e., longitude and latitude) while the vehicle is driving straight. Figure 18a shows the attack (red star). As depicted, right after the attack, the compromised measurements (red curve) is replaced with the software sensor (blue curve), allowing the vehicle to continue its intended trajectory (i.e., straight line). A video is available at [14].

In the *controlled* GPS attack, the attacker maliciously crafts the injected signal based on her/his estimation of the rover's current physical states. In this case, the rover performs a more complex mission in which it moves straight, makes a sharp turn to the left and then drives straight again. During the first straight movement, the attacker injects a signal that closely follows the estimated trajectory but has a small constant error, which misdirects the rover gradually. Specifically, we inject the estimated longitude values with around 0.00001 degrees error, making the attack quite stealthy as it cannot be directly detected. Under this scenario, the vehicle's controller thinks that the vehicle is deviating from the planned mission as the sensed positional measurement are slightly incorrect. To correct the error, the vehicle adjusts its behavior by slightly

moving right. To avoid detection, the attacker makes sure that the error is less than our threshold by accurately estimating the changing real states according to the planned move, and also the effect of the attack. However, during the next maneuver (sharp turn), which is unknown to the attacker, she fails to precisely estimate the real states. Consequently, our recovery monitoring successfully detects the attack and activates the software sensor. Figure 18b shows the planned trajectory (black dotted line), the actual trajectory (black line), the measurements (red curve) used in the controller, the software sensor measurements (blue curve), the attack (red star) and recovery (green star). As shown, during the straight move, the attacker maintains small error by estimating the planned states and the vehicle moves right gradually. However, during the turn, the attack is detected. In this case, the software sensor stops the drift to the right, but it cannot compensate the error introduced by the lengthy attack. In practice, additional emergency steps can be taken, such as reboot or estimate GPS location through external channel (e.g., surroundings and nearby RVs). A video of this attack and recovery is available at [13].

5 Discussion

Recovery Duration. The drift during recovery is inevitable since the software sensor is an approximation. Although our recovery technique successfully prevents sensor attacks temporarily, software sensors cannot replace physical sensors permanently or for a prolonged duration due to the drift effect. Our experiment in Section 4.2 shows that the operation time of the recovery mode can be at least 10 seconds in most cases, enough for launching emergency operations (e.g., emergency landing, manual mode switch with an alert) to avoid devastating incidents. The empirical studies in [17, 23] shows that the takeover time to resume control from a highly automated vehicle is around 3 to 7s. Our recovery duration is 10 seconds.

Advanced Attack. Our experiment in Section 4.3 shows that a small-error attack (carry-off attack) can be detected and prevented. The first reason is that the attacker has no access to the internal sensor readings. However, if the attacker can precisely model such readings, she/he may be able to manipulate the error in a way to evade our defense. For example, if the navi-

gation plan of an RV is extremely simple (e.g., straight-line), the attacker can estimate the internal GPS reading even after it is contaminated by the attack through observing the RV's velocity and considering the accumulated errors introduced so far, and accordingly applies the attack continuously until the goal is reached. One way to defend against this is to avoid any predictable navigation plan, for example, by proactively maneuvering the RV in a specific and secret way unknown to the attacker. Second, our detection mechanism utilizes historical error changes rather than an instant error. This approach can limit the stealthy attack. Specifically, the error between real and model states are calculated with accumulated deviation during a certain time duration. This is distinct from a simple bad-data detector or estimator.

6 Related Work

Our approach is inspired by both traditional hardware and software fault-tolerant techniques. The traditional redundancy-based approach [29] can recover a system when *less than half* of the components have a failure. Moreover, hardware replication requires additional hardware costs. There has been a lot of work regarding physical attacks on RVs in recent years. Many external attacks [8, 47, 49, 51, 52, 54] have been proposed. At the same time, corresponding attack detection techniques [5, 20, 22, 26, 28, 36, 37, 57] have been proposed. However, they focus only on attack detection (i.e., significant anomalies) and do not provide a recovery mechanism for continuous operations. As such, the RV may still crash even though it detects the attack.

State estimation [39] has been well researched in control engineering, aiming to improve the accuracy of noisy sensors. Especially, secure state estimation [19, 38, 48] was introduced to handle sensor attacks. However, it mostly utilizes the remaining benign sensors or sensor redundancy to securely estimate system states in the presence of significant noises or attacks. They restrict attackers to corrupt only a subset of sensors in which case the estimation needs to rely only on the benign sensors. In comparison, our approach uses software sensor based on system modeling regardless of the set of sensors under attack, which is practical and generic.

System identification [32] is used to detect attacks [5] and debug RV failures [56]. Similar to our method, they build models for RVs with SI. However, they only detect extreme deviations and cannot provide accurate feedback to the control loop after detection. In addition, we precisely model individual sensor readings while they cannot.

7 Conclusion

We propose a novel sensor attack recovery technique for multi-sensor RVs. The technique uses generic state-space model based software sensors as a safe backup of the physical sen-

sors. Software sensors can precisely predict physical sensor readings while they are largely isolated from the (malicious) environment. Evaluation with real RVs demonstrates our technique can recover from single and multi-sensor attacks.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported, in part, by ONR under Grant N00014-17-1-2045. Any opinions and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

- [1] Drone - 3DR Solo, 2017. <https://www.3dr.com>.
- [2] ArduPilot :: Home, 2010. <http://ardupilot.org/>.
- [3] Brian Bradie. *A friendly introduction to numerical analysis*. Pearson Education India, 2006.
- [4] Richard R Brooks and Sundararaja S Iyengar. *Multi-sensor fusion: fundamentals and applications with software*. Prentice-Hall, Inc., 1998.
- [5] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. Detecting attacks against robotic vehicles: A control invariant approach. In *Proceedings of the ACM CCS*, pages 801–816. ACM, 2018.
- [6] Self-driving cars now legal in California, 2012. <https://cnn.it/2ZJDnEN>.
- [7] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security, 1998*.
- [8] Drew Davidson, Hao Wu, Robert Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling uavs with sensor input spoofing attacks. In *WOOT*, 2016.
- [9] Case study1: Multiple gyroscope sensor (1,2,3) recovery. <https://youtu.be/SQ1liFlnCB4>.
- [10] Case study2: GPS sensor attack. <https://youtu.be/e8J6ixvtXqQ>.
- [11] Gyroscope sensor attack and crash. https://youtu.be/_b0Lxzvvyu5c.
- [12] Multiple gyroscope sensor (1,2,3) recovery. <https://youtu.be/1MidmM1DEMo>.
- [13] Rover controlled GPS attack. <https://youtu.be/gu1NVssiJls>.

- [14] Rover random GPS attack. <https://youtu.be/S04iL8diRh8>.
- [15] Case study2: GPS sensor recovery, 2019. https://youtu.be/oC3SOGT_XDY.
- [16] W Elmenreich. Sensor fusion in time-triggered systems phd thesis. *Institut für Technische Informatik, Technischen Universität Wien*, 2002.
- [17] Alexander Eriksson and Neville A Stanton. Takeover time in highly automated vehicles: noncritical transitions to and from manual control. *Human factors*, 59(4):689–705, 2017.
- [18] Rover Home — Rover documentation, 2016. <http://erlerobotics.com/blog/erle-rover/>.
- [19] Hamza Fawzi, Paulo Tabuada, and Suhas Diggavi. Secure estimation and control for cyber-physical systems under adversarial attacks. *IEEE Transactions on Automatic control*, 59(6):1454–1467, 2014.
- [20] Paul M Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy: A survey and some new results. *automatica*, 26(3):459–474, 1990.
- [21] Gene F Franklin, J David Powell, Abbas Emami-Naeini, and J David Powell. *Feedback control of dynamic systems*, volume 3. Addison-Wesley Reading, MA, 1994.
- [22] Wei Gao and Thomas H Morris. On cyber attacks and signature based intrusion detection for modbus based industrial control systems. *The Journal of Digital Forensics, Security and Law: JDFSL*, 9(1):37, 2014.
- [23] Christian Gold, Daniel Damböck, Lutz Lorenz, and Klaus Bengler. “take over!” how long does it take to get the driver back into the loop? In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 57, pages 1938–1942. SAGE, 2013.
- [24] Jingqing Han. From pid to active disturbance rejection control. *IEEE transactions on Industrial Electronics*, 56(3):900–906, 2009.
- [25] Pavel Holoborodko. Smooth noise robust differentiators.
- [26] Khurum Nazir Junejo and Jonathan Goh. Behaviour-based attack detection and classification in cyber physical systems using machine learning. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, pages 34–43. ACM, 2016.
- [27] Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Bol. soc. mat. mexicana*, 5(2), 1960.
- [28] Sanmeet Kaur and Maninder Singh. Automatic attack signature generation systems: A review. *IEEE Security & Privacy*, 11(6):54–61, 2013.
- [29] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Elsevier, 2010.
- [30] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. pages 145–159, 2013.
- [31] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
- [32] L Ljung. System identification-theory for the user, prentice hall, upper saddle river n. *System identification: Theory for the user. 2nd ed. Prentice Hall, Upper Saddle River, NJ.*, 1999.
- [33] Kevin M. Lynch and Frank Chongwoo Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.
- [34] System Identification Toolbox - MATLAB, 2017. <https://www.mathworks.com/products/sysid.html>.
- [35] Micro Air Vehicle Communication Protocol, 2017. <http://qgroundcontrol.org/mavlink/start>.
- [36] Robert Mitchell and Ray Chen. Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(5):593–604, 2014.
- [37] Robert Mitchell and Ray Chen. Behavior rule specification based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing*, 12(1):16–30, 2015.
- [38] Yilin Mo and Bruno Sinopoli. Secure estimation in the presence of integrity attacks. *IEEE Transactions on Automatic Control*, 60(4):1145–1151, 2015.
- [39] Katsuhiko Ogata and Yanjuan Yang. *Modern control engineering*, volume 4. Prentice hall India, 2002.
- [40] Tom O’Haver. A pragmatic introduction to signal processing. *University of Maryland at College Park*, 1997.
- [41] Young-Seok Park, Yunmok Son, Hocheol Shin, Dohyun Kim, and Yongdae Kim. This ain’t your dose: Sensor spoofing attack on medical infusion pump. In *WOOT*, 2016.
- [42] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*.
- [43] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*, pages 302–317. Springer, 2016.
- [44] Pixhawk, 2019. <https://pixhawk.org/>.
- [45] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

- [46] Qikun Shen, Bin Jiang, Peng Shi, and Cheng-Chew Lim. Novel neural networks-based fault tolerant control scheme with fault alarm. *IEEE transactions on cybernetics*, 44(11):2190–2201, 2014.
- [47] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 55–72. Springer, 2013.
- [48] Yasser Shoukry, Pierluigi Nuzzo, Alberto Puggelli, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, and Paulo Tabuada. Secure state estimation for cyber-physical systems under sensor attacks: A satisfiability modulo theory approach. *IEEE Transactions on Automatic Control*, 62(10):4917–4932, 2017.
- [49] Yunmok Son, Hocheol Shin, Dongkwan Kim, Young-Seok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, Yongdae Kim, et al. Rocking drones with intentional sound noise on gyroscopic sensors. In *USENIX Security*, pages 881–896, 2015.
- [50] First passenger drone makes its debut at CES, 2016. <https://bit.ly/200zYft>.
- [51] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM CCS*, pages 75–86. ACM, 2011.
- [52] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *EuroS&P*, pages 3–18. IEEE, 2017.
- [53] Zhengbo Wang et al. SONIC GUN TO SMART DEVICES. 2017.
- [54] Jon S Warner and Roger G Johnston. A simple demonstration that the global positioning system (gps) is vulnerable to spoofing. *Journal of Security Administration*, 25(2):19–27, 2002.
- [55] Bin Yao and Chang Jiang. Advanced motion control: from classical pid to nonlinear adaptive robust control. In *AMC*, pages 815–829. IEEE, 2010.
- [56] Enyan Huang Qixin Wang Yu Pei Haidong Yuan Zhijian He, Yao Chen. A system identification based oracle for control-cps software fault localization. In *Proceedings of ICSE*. ACM, 2019.
- [57] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *Proceedings of ICCPS*, pages 109–118. ACM, 2010.

Appendix

A Quadrotor Model and Frames

The Figure 19 shows a quadrotor with two frames: inertia and body frame. The linear position of the quadrotor is defined in



Figure 19: Inertial and body frame

the inertial frame with ξ (x, y, z). The attitude (i.e., angular position) is defined in the inertial frame with η (ϕ, θ, ψ). The roll(ϕ), pitch(θ), yaw (ψ) angle (i.e., Euler angle) determine the rotational angles around the x, y, z axis, respectively. The origin of the body frame is defined in the center of mass of the quadcopter. The linear velocities in the body frame is defined with V_B and angular velocities determined by ω (p, q, r). The rotation matrix R from the body frame to the inertial frame is denoted as:

$$R = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\phi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \quad (8)$$

where $S_x = \sin(x)$, $C_x = \cos(x)$. R is orthogonal thus $R^{-1} = R^T$. The R^T is for rotation from the inertial frame to body frame. The transformation matrix W_η for angular velocities from the inertial frame $\dot{\eta}$ to the body frame ω is:

$$\omega = W_\eta \dot{\eta}, \quad \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & C_\theta S_\phi \\ 0 & -S_\phi & C_\phi C_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad (9)$$

Also, the Euler angular velocity is then

$$\dot{\eta} = W_\eta^{-1} \omega, \quad \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & S_\phi T_\theta & C_\phi T_\theta \\ 0 & C_\phi & -S_\phi \\ 0 & S_\phi / C_\theta & C_\phi / C_\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}, \quad (10)$$

B Supplementary Compensation.

We present an example of the supplementary compensation approach that estimates angle status from the accelerometer and magnetometer. Note that angle status is typically measured by gyro. The conversion equation is the following.

$$\begin{aligned} \phi_{acc} &= \text{atan2}(y_{acc}, \sqrt{x_{acc}^2, z_{acc}^2}) \\ \theta_{acc} &= \text{atan2}(x_{acc}, \sqrt{y_{acc}^2, z_{acc}^2}) \\ \psi_{mag} &= \text{atan2}(-y_{mag} \cdot \cos\phi + z_{mag} \cdot \sin\phi, \\ &\quad x_{mag} \cdot \cos\theta + y_{mag} \cdot \sin\theta \cdot \sin\phi + z_{mag} \cdot \sin\theta \cdot \cos\phi) \end{aligned} \quad (11)$$

The conversion errors and the real sensor errors (e.g. sensor noise, bias) cause fluctuations in the equations' output. We use low-pass filter to smooth the outputs. Combining the outputs from Eq. (11) and our software sensor (using weighted sum), we acquire more accurate measurements.

SIEVE: Secure In-Vehicle Automatic Speech Recognition Systems

Shu Wang¹, Jiahao Cao^{1,2}, Kun Sun¹, and Qi Li^{2,3}

¹Center for Secure Information Systems, George Mason University, Fairfax, VA, USA

²Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China

³Beijing National Research Center for Information Science and Technology, Beijing, China

Abstract

Driverless vehicles are becoming an irreversible trend in our daily lives, and humans can interact with cars through in-vehicle voice control systems. However, the automatic speech recognition (ASR) module in the voice control systems is vulnerable to adversarial voice commands, which may cause unexpected behaviors or even accidents in driverless cars. Due to the high demand on security insurance, it remains as a challenge to defend in-vehicle ASR systems against adversarial voice commands from various sources in a noisy driving environment. In this paper, we develop a secure in-vehicle ASR system called SIEVE, which can effectively distinguish voice commands issued from the driver, passengers, or electronic speakers in three steps. First, it filters out multiple-source voice commands from multiple vehicle speakers by leveraging an autocorrelation analysis. Second, it identifies if a single-source voice command is from humans or electronic speakers using a novel dual-domain detection method. Finally, it leverages the directions of voice sources to distinguish the voice of the driver from those of the passengers. We implement a prototype of SIEVE and perform a real-world study under different driving conditions. Experimental results show SIEVE can defeat various adversarial voice commands over in-vehicle ASR systems.

1 Introduction

Driverless cars, also known as autonomous cars or self-driving cars, are no longer the things we would only see in sci-fi films, but they are becoming an irreversible trend in our daily lives, particularly due to the rapid development of sensor techniques and advanced artificial intelligence algorithms. For instance, MIT developed a human-centered autonomous vehicle using multiple sensors and deep neural networks in 2018 [1]. From April 2019, all new Tesla cars come standard with Autopilot, providing the capability of switching modes between manual driving and self-driving [2]. Waymo's driverless cars have driven 20 million miles on public roads by January 2020 [3].

The latest in-vehicle voice control system [4] provides a

convenient way for drivers and passengers to interact with driverless cars. For example, we can use our voice to control in-vehicle entertainment systems, set destinations to the GPS navigation system, and take back the full control of cars from the self-driving mode [5]. However, the core module of in-vehicle voice control system, i.e., automatic speech recognition (ASR) module, is vulnerable to various adversarial voice command attacks [6–14]. Particularly, since most in-vehicle ASR systems support speaker-independent recognition by default [15], passengers can voice malicious commands to ASR systems and thus control critical in-vehicle systems. Moreover, remote attackers may hide a voice command into a song [9]. When the song is played through car loudspeakers or smartphones' speakers, the malicious voice command in the song can be recognized by ASR systems. It may cause unexpected behaviors or even accidents in driverless cars.

A number of countermeasures [16–18] have been proposed to defend ASR systems against adversarial voice commands. They leverage short-term spectral features [19,20] or prosodic features [21,22] to distinguish different users. However, as the features of human voices typically are low dimensional, advanced passengers can imitate a driver's voice to bypass existing defense systems [23]. Moreover, existing methods on distinguishing different users' identities are generally unreliable in a noisy environment [24], whereas in-vehicle ASR systems demand a much higher security insurance against adversarial voice command attacks to prevent possible car accidents. Furthermore, to prevent malicious voice commands played by speakers, researchers have designed methods to identify whether the voice commands come from humans or loudspeakers. They rely on spectral features [18,25,26] and noise features [17,27] to distinguish between humans and speakers. They are all based on one underlying assumption that voice commands come from a single source. However, in the scenario of driverless cars, malicious voice commands may come from multiple sources (i.e., multiple car speakers). Therefore, the different features for multi-source voices and single-source voices may interfere with distinguishing between human voices and non-human voices [28].

In this paper, we propose a secure automatic speech recognition system called SIEVE to effectively defeat various adversarial voice command attacks on driverless cars. It is capable of distinguishing voice commands issued from a driver, a passenger, and non-human speakers in three steps. First, since legal human voice commands are always single-source signals, SIEVE identifies and filters out multiple-source voice commands from multiple car speakers. The multiple-source detection is based on a key insight that when the same signal is received multiple times in a short time period from multiple sources, the overlap of the received signals will expand the signal correlations in the time domain. Therefore, SIEVE can identify multi-source voice commands by conducting an autocorrelation analysis to measure the overlap of signals.

After filtering out multiple-source voice commands, the second step of SIEVE is to check if a single-source voice command is from a human or a non-human speaker. SIEVE can accurately detect non-human voice commands by checking features in both frequency domain and time domain. First, voices from non-human speakers inherently have the unique acoustic characteristic, i.e., low-frequency energy attenuation. Such characteristic can be checked with the signal power spectrum distribution in the frequency domain. However, sophisticated attackers may use low-frequency enhancement filters to modulate the voice and thus compensate for the energy loss. To identify such modulated voices, SIEVE also conducts a local extrema verification in the time domain. Our key insight is that the local extrema ratio for modulated voices is much greater than that for human voices. Moreover, we demonstrate that attackers cannot modulate voices to bypass the detection in both the time domain and frequency domain at the same time. Hence, our dual-domain check ensures SIEVE can effectively filter out various non-human voice commands.

Finally, SIEVE distinguishes the passenger's voice commands from the driver's voice commands, since we may only trust the driver but not the passengers. Our key insight is that vehicles have fixed internal locations for the driver and passengers. Therefore, we can leverage the directions of voice sources to distinguish the driver's voice and passengers' voice even if passengers can imitate the driver's voice. Particularly, SIEVE measures the directions of voice sources by calculating the time difference of arrivals (TDOA) on a pair of close-coupled microphones (i.e., a dual microphone). To deal with some extreme cases when passengers lean forward and have their head near the headrest of the driver's seat, we also develop a spectrum-assisted detection method to improve the detection accuracy of SIEVE.

We implement a prototype of SIEVE and conduct extensive real-world experiments on a four-seat sedan (Toyota Camry) under various vehicle driving states, including idling, driving on local streets, and driving on highway. The experimental results show that our system can effectively defeat adversarial voice command attacks. For example, SIEVE can achieve a 96.75% accuracy on distinguishing human voices from non-

human voices when the car is driving in noisy streets. It can further identify the driver's voice from human voices with a 96.76% accuracy. Moreover, our system can be smoothly integrated to vehicles by replacing the in-vehicle single microphone with a low-priced dual microphone and implanting the detection module in one vehicle electronic control unit.

In summary, we make the following contributions:

- We develop a secure ASR system for driverless vehicles to defeat various in-vehicle adversarial voice commands by distinguishing the command sources from the driver, passengers, and electronic speakers.
- We propose a dual-domain detection method to distinguish voice commands between humans and non-human speakers even if the voices are carefully modulated to mimic human voices.
- We provide a method based on the directions of voice sources to distinguish the driver's voice from passengers' voices even if passengers can imitate the driver's voice.
- We implement a prototype of SIEVE and real-world experimental results show that our system can effectively defeat adversarial voice commands.

2 Threat Model and Assumptions

We focus on the adversarial voice command attacks that manipulate the speech inputs to the in-vehicle ASR system. We assume the vehicle's electronic control unit (ECU) can be trusted [29]. We assume the driver can be trusted to not issue malicious commands; however, malicious voice commands may be issued from in-vehicle loudspeakers, the speakers of mobile devices, or passengers.

First, malicious voice commands may come from the in-vehicle loudspeakers. It is common for people to connect their mobile phones to car audio systems when playing music or making phone calls. Also, CDs/DVDs are usually played through speakers. Since music songs might be downloaded from various untrusted sources, the attackers may edit the sound tracks of audio files to voice the malicious commands via a single speaker or multiple speakers. Particularly, armored attackers may hide adversarial commands by minimizing the difference between the malicious and the original audio samples [7–9]. Moreover, when a phone call is connected to the vehicle speakers via Bluetooth, the people on the other side may unintentionally or intentionally issue voice commands to the ASR systems.

Second, if the driver puts their smartphones on mobile speakers (handsfree mode) when making phone calls or playing musics, a malicious command may be issued from the driver's smartphones. Similarly, a passenger's smartphone may be exploited to voice malicious commands. Thus, it is necessary to identify the voice commands issued from the speakers of mobile devices such as smartphones.

Third, passengers may issue dangerous or annoying human voice commands to ASR systems. For instance, kids

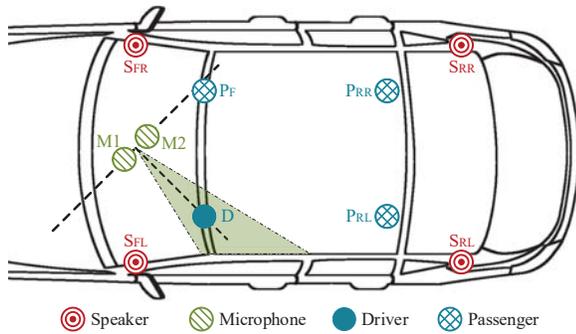


Figure 1: Car Internal Structure and the Location and Orientation of a Dual Microphone.

may unintentionally voice wrong commands to the vehicle or intentionally mess with the recreation systems. Moreover, we assume malicious passengers may bring or leave some dedicated portable hardware to launch advanced attacks such as those in an inaudible frequency range to humans [10–12] (though the sizes of most dedicated hardware devices may not be small). It is critical to distinguish the voice commands from the passengers or their dedicated mobile devices.

3 System Design

We first provide an overview of our system and then present the detailed techniques used in each detection step.

3.1 System Overview

Figure 1 shows the typical internal structure of a four-door sedan, which has at least four speakers installed in four corners (front/rear and left/right) to achieve good stereophonic experience. It has four seats for the driver (D), the front passenger (PF), the rear left passenger (PRL), and the rear right passenger (PRR).

The entire defense system consists of three detection steps. The first step is to identify and filter out the voices coming from multiple speakers since human’s legal commands are issued from single voice source. The second step is to detect adversarial voice attacks from loudspeakers (i.e., replay attacks), no matter the in-vehicle speakers or mobile speakers. The third step is to identify the voice source from its direction by using a dual microphone in the front of the sedan. This step can distinguish voices from the driver and any passenger.

Step 1: Detecting Voice from Multiple Speakers. When attackers use multiple vehicle speakers to perform voice command attacks, the reverberation effect is enlarged since a microphone captures the same signals multiple times at different moments. Since the overlapping of multiple copies of the same signals within a small time expands the signal correlations in the time domain, the linear prediction (LP) residual [30] of the signal can be calculated to decide if the voice commands are received from multiple speakers. Moreover, Hilbert envelope and local enhancement techniques are

used to enhance the significant excitation. The basic idea is that the relative time delays of the instants of significant excitation remain unchanged in the audio signals captured by the microphones. Therefore, through accumulating the auto-correlation results over the entire voice command signal, we can compare the different patterns to distinguish single-source signals from multi-source signals. Comparing with other methods [28, 31, 32], we adopt the LP residual method since it can achieve a higher detection precision.

Step 2: Distinguishing between Human Voice and Voice from Single Speaker. We develop two new approaches, namely, power spectrum verification and local extrema cross-check, to detect voice from an electronic speaker. Since the common speakers can suppress the power of the low-frequency signals, we use the power spectral density to distinguish the human voice from the replay voice. To escape our power spectrum checking, the attackers may design an inverse filter to compensate the speaker’s frequency response. We can defeat this armored attack by performing a local extrema verification in the time domain. By combining these two checks on both frequency-domain features and time-domain features, we can accurately detect the voice coming from loudspeakers.

Step 3: Distinguishing Driver from Passengers. We use a dual microphone to decide the direction of the voice commands. The dual microphone consists of a pair of microphones (M_1 and M_2) that are close to each other (e.g, 5 centimeters). When a voice is captured by the two microphones, we use a far-field model to measure the time difference of arrivals (TDOA) [33] since the source-microphone distance is much larger than the distance between these two microphones. To maximize the detection accuracy, we orient the dual microphone in a direction that the line connecting the two microphones is perpendicular to the line connecting the driver seat and the middle point of the two microphones, as shown in Figure 1. The cross-correlation function of the two-channel signals is effective on measuring the time delay between two channels. As shown in Figure 2, the angle range (inside the vehicle) can be divided into multiple small pie regions, since the TDOA measured value and the propagation angle follow a *arccosine* function (see details in Section 3.4). When the voice propagation direction is perpendicular to these two microphones, the measurement can achieve the highest precision on recognizing the driver. It is the reason why the dual microphone is oriented in Figure 1.

With the orientation of the dual microphone, when a voice comes from the driver’s direction, the cross-correlation function is almost central symmetric due to the negligible time delay between two signal channels. When the voice comes from any passenger, the cross-correlation function would be left skewed. In Figure 2, the gray areas represent identification regions for different passengers. In most cases it can accurately distinguish the driver’s direction from those of passengers, and we confirm it in our real-world experiments. In some cases, it is challenging to distinguish the driver from

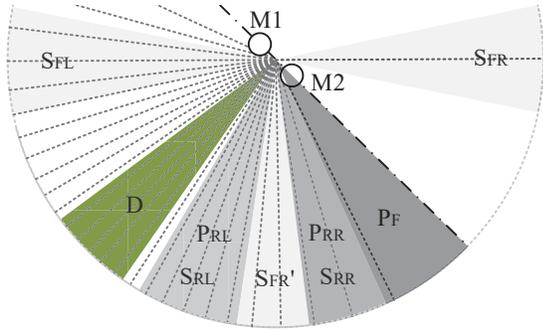


Figure 2: Voice Source Directions to Dual Microphone.

the passenger sitting behind the driver (i.e., P_{RL}), particularly, when the driver may lean towards the right side of the driver’s seat (e.g., resting their arms on car armrests) and the passengers may lean forward and have their head near the headrest of the driver’s seat. We develop a spectrum-assisted detection technique that combines the location of a particular voice with the voice’s specific spectrum features to improve the detection accuracy.

3.2 Detecting Multiple Speakers

The basic idea of the multiple speaker detection is that the reverberation effect occurs since the microphones will capture the same signal multiple times at different instants.

Signal Representations. In the time domain, the captured signals $x(n) = \sum_{i=1}^m A_i \cdot s(n - N_i)$ are the overlapping of several time-shifted signals with different attenuation coefficients, where $s(n)$ is the original signal, N_i and A_i denote the time delay and the attenuation coefficient of the i -th signal, and m is the number of speakers. The different time delays depend on the relative locations between microphones and speakers, which are fixed in vehicles. The only difference between the signals captured by two microphones is a slight time shift related to the distance of microphones, so we only need to process the captured signal in any one of the two microphones.

The time delays can be extracted from the significant excitation regions with high signal-noise ratio (SNR). However, the noise will adversely affect the extraction of the time delays. Therefore, it is necessary to reduce the noise and amplify the strong excitations. Also, to reduce the second-order correlations and inhibit the reflection, we extract the linear prediction residuals from the captured signals. According to the parameter estimation model [30], the linear prediction residuals are obtained as $e(n) = x(n) + \sum_{k=1}^p a_k \cdot x(n - k)$, where a_k is the predictor coefficients and p is the order of the prediction filter.

Signal Enhancements. The peaks in the linear prediction residuals are of double polarity, which introduces fluctuations to the autocorrelation function. For convenient calculation, linear prediction residuals could be converted into a single polarity form by the Hilbert envelope [34], denoted as $h(n) = \sqrt{e^2(n) + e_h^2(n)}$ where $e_h(n)$ is the Hilbert transform of $e(n)$.

The Hilbert envelop signal can describe the amplitude change of the original signal. However, the weak peaks in the Hilbert envelop may lead to spurious high values in the auto-correlation calculation. Thus, we adopt the local enhancement method to further highlight the high SNR regions. We set a sliding window to calculate the signal mean value in the local area. Then, the Hilbert envelop can be enhanced by taking the square of the original signal over the local signal mean as

$$g(n) = \frac{h^2(n)}{\frac{1}{2M+1} \sum_{k=n-M}^{n+M} h(k)}, \quad (1)$$

where $g(n)$ is a preprocessed signal of Hilbert envelope and $(2M + 1)$ denotes the length of the sliding window.

Autocorrelation Analysis. When taking a L -length segment in $g(n)$ as a reference, the autocorrelation function $C(s)$ of the signal $g(n)$ can be calculated as

$$C(s) = \frac{\sum_{k=N}^{N+L-1} g(n) \cdot g(n+s)}{\sqrt{\sum_{k=N}^{N+L-1} g^2(n) \cdot \sum_{k=N+1}^{N+L-1+s} g^2(n)}}, s \in [-S, S], \quad (2)$$

where N is the start index and L is the segment length. The autocorrelation value is normalized by the square mean of the segmented signal. The autocorrelation function is calculated over the interval $[-S, S]$. S indicates the maximum detection range that should be larger than the maximum time delay.

$$S > \frac{D_{max} - D_{min}}{v_0} \cdot f_s = \frac{\Delta D}{v_0} \cdot f_s, \quad (3)$$

where D_{max} and D_{min} indicate the maximum and minimum distance between speakers and microphones, and v_0 is the speed of sound in air with a typical value of 345 m/s. f_s is the sampling rate of microphones. Only if S is larger than the maximum possible time delay, the autocorrelation function can record all the time delays information for the speakers.

Judgment Criteria. The autocorrelation function will have several peaks that correspond to the time delays between different propagation paths. For accurate estimation, we judge the results by multiple signal segments rather than a single one. For each autocorrelation function, we only extract the most significant peak that corresponds to the most distinct time delay. In the i -th signal segment, the offset of the highest peak in the autocorrelation function is denoted as $p_i = \arg \max C_i(s)$. To reduce the effects of noise and spurious peaks, we acquire the statistical distribution of p_i with the autocorrelation functions in multiple signal segments.

For a single voice source, most of p_i values are close to zero, resulting in a concentrated distribution. For multiple voice sources, the p_i distribution is rather dispersed. That is because the voices come from different speakers have different arrival moments, resulting in large time delays in the captured signals. Based on these attributes, we can distinguish the patterns between multiple-speaker signals and single-speaker signals according to the dispersion of the p_i distribution. The dispersion P is measured as the proportion of p_i in the interval

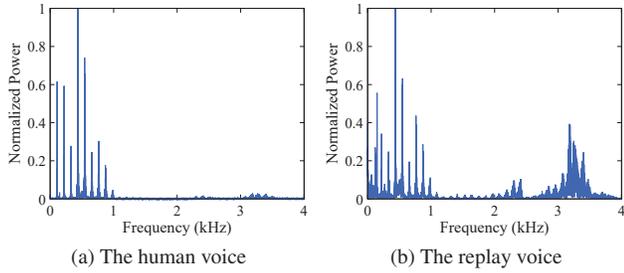


Figure 3: Power Spectral Density for Voice Signals.

$[-d, d]$, where d is a small number compared with S . λ is the decision threshold that obtained from multiple experiments. If $P \geq \lambda$, it means the time delays are centered around 0 and the voice comes from a single source. If $P < \lambda$, the voice comes from multiple sources due to the dispersed distribution. Since the voices from multiple sources are most likely adversarial voices generated by attackers, we can safely filter out these multi-source voices in the first step.

3.3 Identifying Human Voice

After verifying a voice coming from a single source, we detect and filter out the voice that comes from loudspeakers. We solve this challenge by combining two approaches, namely, *frequency-domain power spectrum verification* and *time-domain local extrema cross-check*, to ensure that the voice indeed comes from humans.

Frequency Domain Verification. This approach is based on a noticeable timbre difference between a human voice and a replay voice sound from loudspeakers. Human beings voice commands through the phonatory organ, resulting in a sound frequency typically from 85 Hz to 4 kHz [35]. However, a dynamic loudspeaker can suppress the signals in the low-frequency range due to the limited size, especially under the frequency of 500 Hz [36]. Thus, even a speaker replays a recorded human voice that contains the same frequency components, the timbre is totally different from the genuine one. The main reason is that different power distributions of frequency components lead to different timbre [37].

By leveraging the characteristic of different power distributions, we can distinguish the voice coming from a human or a loudspeaker. The captured voice will be verified with the power spectral density, specifically the ratio of the low-frequency power. The frequency of human voice ranges from 85 Hz to 4 kHz, among which the low-frequency components are dominant, as shown in Figure 3(a). The replay audio sound from loudspeaker has the similar frequency components; however, the sharp decrease in the low-frequency components increases the relative ratio of the high-frequency components, as shown in Figure 3(b). In our design, a voice that comes from a single source is further verified by evaluating the power ratio of the low-frequency components (85 Hz - 2 kHz) to all frequency components. If the voice indeed comes from humans,

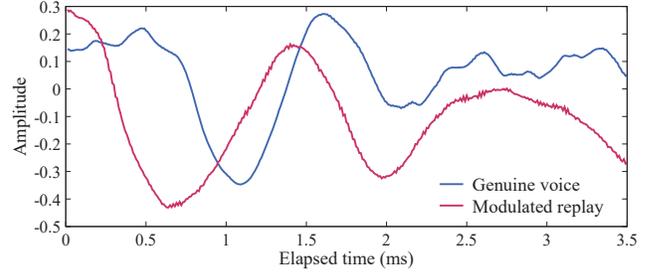


Figure 4: Time-domain Waveform for Voice Signals.

the power ratio should be greater than a specific threshold.

However, an ingenious attacker may compensate the loss of the low-frequency energy by modifying the recording file in the frequency domain. By estimating the transmission properties $K(f)$ of the loudspeakers, an attacker can design an inverse filter $K^{-1}(f)$, where $K(f) \cdot K^{-1}(f) = 1$. Then the attacker can reconstruct the audio file with $K^{-1}(f)$ for compensating the speaker frequency response, and we call such voice as *modulated replay voice*. As the frequency response of the loudspeaker and the inverse filter cancel each other during playback, it is difficult to solely rely on the frequency-based method for distinguishing the modulated replay voice from the genuine voice. Fortunately, we could combine a verification approach in the time domain.

Time Domain Verification. We observe there are different patterns in the local extrema ratio of the human voice and the modulated replay voice. In a 3-length window of time-domain signal, if the midpoint is the maxima or minima in the window, we define the midpoint as a *local extrema* [38]. Also, the ratio of the local extrema amount to the total signal length is defined as *local extrema ratio*. Though the local extrema are not directly related to the spectrum, the number of local extrema can indirectly reflect some spectrum features.

The attacker can only compensate the voice signals with the amplitude spectrum. The phase spectrum is hard to be compensated because of the difficulty to measure the speaker phase response. Due to the phase mismatch errors in the modulated voice, the time-domain signal will contain extremely small oscillation, namely ringing artifacts (see Figure 4). These artifacts cannot be heard by a human, but the local extrema ratio of the modulated replay voice is much greater than that of the human voice in the time domain.

Because the local extrema ratios of the human voice and the modulated replay voice are different, we can identify if the voice indeed comes from a human or a loudspeaker by combining both the frequency-domain power spectrum verification and the time-domain local extrema cross-check. A verified human voice command must satisfy two conditions: (1) the low-frequency power dominates; (2) it complies with the human voice patterns in local extrema ratio. It is difficult for attackers to meet both requirements by manipulating the limited-sized loudspeakers.

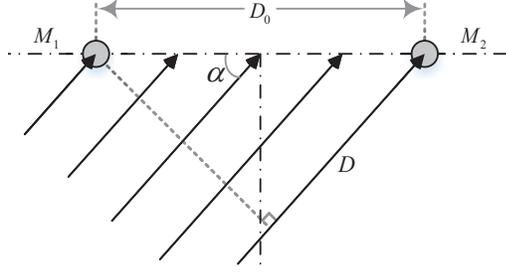


Figure 5: Principle of Time Difference of Arrival.

3.4 Identifying Driver's Voice

In the third step, we identify the voice source via the voice propagation direction and thus distinguish the commands voiced from the driver or any passenger.

Time Difference of Arrival. Two spatially separated microphones can detect the voices from different propagation directions using the time difference of arrival (TDOA), as shown in Figure 5. The distance between M_1 and M_2 is denoted as D_0 . Since the distance between voice source and microphones is larger than the distance between two microphones, we use a far-field model [39] to calculate the time difference of arrivals. The angle between the voice propagation direction and M_1M_2 is denoted as α . If α is 0 or π , the propagation direction is parallel to M_1M_2 , resulting in the largest difference of arrival. If α is equal to $\pi/2$, the propagation direction is perpendicular to M_1M_2 , and the difference of arrival will be zero. Due to the microphone placement shown in Figure 1, zero difference of arrival means the voice comes from the driver.

In Figure 5, D is the difference of the propagation distances between the voice source and two microphones. Thus, the time difference of arrivals can be calculated as $\Delta t = D/v_0 = (D_0 \cdot \cos \alpha)/v_0$. Because the voice is recorded as a digital signal, the captured voice is discrete in the time domain. In the captured signal, the difference of arrivals in sampling units ΔN can be estimated as $N_0 - 1 < \Delta N \leq N_0 + 1$, where $N_0 = \Delta t \cdot f_s$. To simplify the calculation, we approximate ΔN as $\Delta N \propto D$. Thus, we can obtain $\Delta N \approx D_0 \cdot \cos \alpha \cdot f_s/v_0$. The propagation angle α that calculated as follows will be used to determine the direction of the voice source.

$$\alpha = \arccos\left(\frac{\Delta N \cdot v_0}{D_0 \cdot f_s}\right). \quad (4)$$

Detection Precision. The effects of the ΔN changes on α are different. When α is approximately equal to $\pi/2$, for each change of one unit in ΔN , the change in α can be calculated according to Taylor series expansion: $\Delta \alpha = x + o(x)$, where $x = v_0/(D_0 \cdot f_s) < 1$. When α approaches 0 or π , for each change of one unit in ΔN , the change in α can be calculated according to Puiseux series expansion: $\Delta \alpha = \sqrt{2x} + o(x)$.

As a result, if $\alpha \approx \pi/2$, for each unit change of ΔN , $\Delta \alpha$ will be less than that one when $\alpha \approx 0$ or π . More units are concentrated around $\alpha \approx \pi/2$. Thus, the system becomes more sensitive to the angle change near the driver's direction, pro-

viding the optimal detection precision for the driver's voice.

Signal Preprocessing. When a sound signal arrives, we first check if it is a usable signal. Since the frequency of speech signal is between 85 Hz (f_l) and 4000 Hz (f_h), a fast Fourier transform (FFT) is used to obtain the frequency spectrum $X(k)$ of the captured signal $x(n)$, where $X(k) = FFT(x(n))$. Then we judge the speech signal by verifying the power ratio of a band-pass signal $R_p = \sum_{k=k_l}^{k_h} X^2(k) / \sum_{k=0}^{K/2} X^2(k) > \epsilon$. K is the amount of points in the FFT. $k_l = \lfloor K f_l / f_s \rfloor$, $k_h = \lfloor K f_h / f_s \rfloor$, where $\lfloor x \rfloor$ means the largest number less than x . $\epsilon = 0.57$ is a threshold value obtained from our experiments. The signal will only be processed in this step when $R_p > \epsilon$, because the higher SNR signal is suitable for the TDOA algorithm [40].

To reduce high-frequency noise and obtain a smooth signal waveform, we process the captured signal with a pre-set low-pass filter in the frequency domain. The smooth voice signal $y(n)$ can be obtained by the inverse fast Fourier transform (IFFT). $y(n) = IFFT(X(k) \cdot H(k))$, where $H(k)$ is a low pass filter that inhibits the high frequency components.

Cross-Correlation Evaluation. According to the TDOA algorithm, the cross-correlation function between two-channel signals is given by the following equation.

$$C_{12}(s) = \sum_{n=n_0}^{n_0+l-1} y_1(n-s) \cdot y_2(n), \quad -S_m \leq s \leq S_m, \quad (5)$$

where n_0 is the start index, l is the segment length, $y_1(n)$ and $y_2(n)$ are the signals captured by M_1 and M_2 . S_m is the max shift value that subjects to the constraint $S_m \geq D_0 \cdot f_s/v_0$.

In the cross-correlation function, the shifted sampling unit with the maximum cross-correlation value indicates the time delay between two channels. The corresponding offset value of the cross-correlation peak is denoted as $s_0 = \text{argmax}(C_{12}(s))$. According to Equation (4), the voice propagation angle can be estimated as $\alpha = \arccos[(s_0 \cdot v_0)/(D_0 \cdot f_s)]$. Note if a voice comes from any passenger, s_0 will be a negative value as the propagation angle α is greater than $\pi/2$.

In Figure 1, the decision criterion is $|\alpha - \pi/2| \leq \alpha_T$ if a signal is recognized as the driver's voice. α_T is an angle threshold that demarcates the decision boundary. Considering the relationship between α and s_0 , we only need to use the decision criterion $|s_0| \leq s_T$ with an offset threshold s_T . If s_0 satisfies the above condition, it means the voice comes from the direction approximately perpendicular to M_1M_2 . Thus, the captured voice can be recognized as coming from the driver.

Spectrum-assisted Detection. In real-world situation, the driver may lean to the right side on the armrest during driving, and its voice may fall into the angle range of the rear-left passenger. To identify the driver's voice more robustly, we develop a spectrum-assisted detection technique to allow the voice of the driver to move within a wider angle range without sacrificing the detection accuracy. The basic idea is to combine specific spectrum characteristics of the wake-up voice command (e.g., "Hi, SIEVE") with the direction of the voice.

To determine the same voice source (i.e., the driver), we record the spectrum histogram and the propagation direction of previous wake-up commands. For the i -th command that has been successfully recognized as the driver’s voice, the m -bar spectrum histogram of the wake-up command is denoted as v_j^i , $j = 1, \dots, m$, and the propagation angle for the i -th voice command is denoted as α^i . For the next $(i + 1)$ -th command, the received wake-up command must satisfy two conditions. First, the spectrum statistics of wake-up commands are similar, indicating the voice commands come from the same person. The spectrum similarity can be measured using the root-sum-square of histogram difference $(\sum_j (v_j^{i+1} - v_j^i)^2)^{1/2} < th_1$, where th_1 is a similarity threshold. Second, the voice movement is within an acceptable wider range (e.g., the driver’s voice cannot come from the seats on the right), which is measured by the angle difference $|\alpha^{i+1} - \alpha^i| < th_2$, where $\alpha^0 = \pi/2$ is the theoretical measured angle of the driver. And the angle threshold th_2 can be $\pi/4$, indicating that the driver sits on the left side of the car. If a newly received wake-up command satisfies both conditions, we would consider the voice command is coming from the driver.

With the spectrum-assisted detection method, our system can successfully recognize the driver’s commands even though the driver is in a different sitting posture. Also, if the person sitting in the seat directly behind the driver leans forward and has his head near the headrest of the driver’s seat, our system can still reject his commands since the commands only satisfy the angle constraint but not both constraints.

4 Experimental Results

In our experiments, a TASCAM DR-40 portable digital recorder with two spatially separated microphones is used to capture the voice signals, as shown in Figure 6(a). The distance between the two receivers D_0 is 5 cm. The sampling rate of both microphones is 96 kHz (f_s). We not only analyze the data captured in the lab environments, but also test our system in real world, as shown in Figure 6(b). The vehicle model is Toyota Camry LE 06 with two Scion TC XB 6.5-inch speakers and two Kicker 43DSC69304 D-Series 6x9-inch speakers. To test more loudspeakers, we use three smartphones (iPhone X, Google Nexus 5, and Xiaomi Mi 4) with their built-in speakers. Since we cannot modify the electronic control unit (ECU) of the vehicle, the system runs in an environment on a laptop with Intel Core i7-7700, 2.8GHz CPU with 16GB RAM.

4.1 Accuracy on Detecting Multiple Speakers

In the multiple speakers detection, the linear prediction residuals are obtained by a 12-order linear prediction filter. A sliding window with 2001 units length is used to enhance the preprocessed signals. The length of the signal segments is 512 units, and the maximum offset value is also 512 units because the maximum distance difference between speakers



Figure 6: Experiment Setup.

and microphones is 1.5 meter in our experiments. According to Equation (3), S should be greater than 417 units. The threshold λ is set as 0.33 through 42 experiments. Under this condition, the total recognition accuracy can reach 83.3%.

We conduct experiments by using different combinations of speakers. The test audio was originally collected by the digital recorder with a sampling rate of 96 kHz. One 4-track audio file with 15 track combinations (4 combinations for 1 speaker, 6 combinations for 2 speakers, 4 combinations for 3 speakers, and 1 combination for 4 speakers) is edited via MATLAB vector operations. The test recording file is finally generated by the *wavwrite* tool [41]. The detection accuracy of a single or four in-vehicle speakers is 100%, while the average accuracy of detecting two and three speakers is 66.7% and 75%, respectively. It is challenging to identify two front speakers or two rear speakers since in that case ΔD is small and easy to ignore. However, when considering the voice from those two speakers as from one source, we still can filter them out according to their directions in the third step.

4.2 Accuracy on Detecting Human Voice

By evaluating the power ratio of low-frequency components, we can distinguish the human voice from the replay voice sound from loudspeakers. In Figure 7(a), 97.3% of human voices have the low-frequency power ratio of over 0.995. The low-frequency power ratio of a replay voice is distributed and less than that of a human voice. Based on these features, we can distinguish if the voice commands sound from the driver or a loudspeaker. In our experiments, the detection accuracy on replay voices is 99.05% with the threshold of 0.96.

We also confirm that the low-frequency power of the modulated replay voice dominates after the artificial enhancement, which makes our frequency-based method unreliable. Therefore, we should also cross-check the voice in the time domain, where the voice pattern can be obtained by calculating the local extrema ratio. The pattern difference of the human voice and the modulated replay voice is illustrated in Figure 7(b). Because of the ringing artifacts, the modulated replay voice has a larger local extrema ratio, typically greater than 35%.

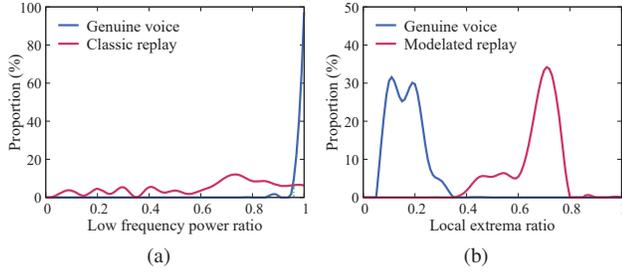


Figure 7: Frequency-domain and Time-domain Features for Detecting Human Voice. (a) the low-frequency power ratios between human voice and replay voice; (b) the local extrema ratios between human voice and modulated replay voice.

While the local extrema ratio of human voice is typically less than 35% due to the statistical smooth.

Therefore, we can successfully distinguish the human voice from the modulated replay voice via the local extrema cross-check methods. When the decision threshold is 0.35, the detection accuracy can achieve 99.62%.

4.3 Accuracy on Detecting Driver's Voice

A series of experiments are conducted to distinguish voices coming from the driver or a passenger. The length of the signal segments is 512 units. The maximum offset S_m is 64, which should be greater than the theoretical maximum peak offset $(\Delta N)_{max} \approx 14$ units.

We first perform experiments with five basic voice propagation angles ($0, \pi/4, \pi/2, 3\pi/4, \pi$) in a quiet lab environment. Figure 8(a) shows the results of cross-correlation in logarithmic form with normalization, verifying the correctness of our theoretical analysis. When the propagation angle α is 0 or π , the absolute offset of the peak $|s_0|$ is 14 units, the same as $(\Delta N)_{max}$. When α is $\pi/4$ or $3\pi/4$, the absolute offset of the peak $|s_0|$ is 10 units, which follows the equation $s_0 \approx (\Delta N)_{max} \cdot \cos(\alpha)$. The peak offset s_0 is near 0 when the voice comes from the direction perpendicular to M_1M_2 . This property enables the system to distinguish the driver's voice and the voice coming from any passenger. Another intriguing property in Figure 8(a) is the different detection precision over various angles. The α changes from 0 to $\pi/4$ are presented by 4 units, while 10 units are used to measure the α changes from $\pi/4$ to $\pi/2$. Higher precision can be achieved near the angle of $\pi/2$, which means we can get the best detection performance in the driver's direction. It explains why we orient two microphones with a 45-degree angle to the vehicle.

We also conduct experiments in a real car to distinguish the driver's voice from passengers' voices. Figure 8(b) shows the cross-correlation functions of the driver's voice and three passengers' voices. The cross-correlation functions of the passengers are left-skewed with negative peak offsets since the propagation angles are greater than $\pi/2$. The driver's voice has a near-zero peak offset due to the propagation angle of

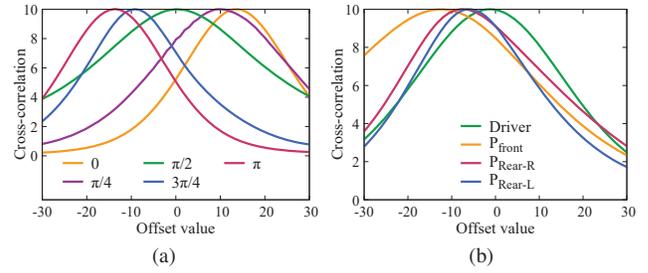


Figure 8: Experimental Results under Different Situations. (a) cross-correlation functions for 5 propagation angles in a quiet lab environment; (b) cross-correlation functions for the driver and three passengers.

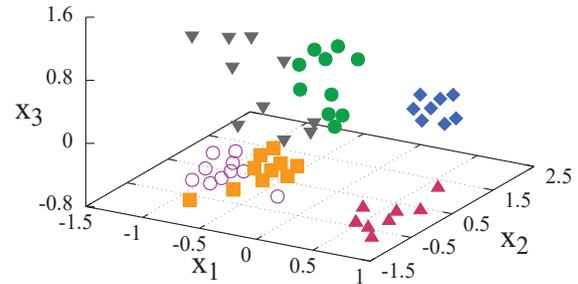


Figure 9: The Spectrum Features of the Wake-up Command for 6 Users in PCA Low-dimensional Subspace.

$\pi/2$. The absolute peak offset of P_{Rear-L} is 6 units, which is the closest to the driver. Thus, according to the decision criterion, SIEVE can distinguish the driver from three passengers with a decision threshold (s_T) of 2. In the experiments, 2437 signal segments are tested and the overall detection accuracy can achieve 96.76%. The false positive rate is 2.86%, while the false negative rate is 4.44%. The statistics show the system is accurate and robust when distinguishing the driver's voice from the passengers' voices.

To verify the validity of the spectrum-assisted detection method, we collect 60 wake-up commands that are issued from 6 different users. Then we utilize a 10-bar spectrum histogram (from 0 to 2 kHz) to extract the spectrum features of the wake-up command. With the similarity threshold th_1 of 0.081, the accuracy of correlating two voice commands can achieve 92.72% (i.e., only 262 out of 3600 pairs are misjudged). After applying the PCA dimension reduction [42], we can visualize the spectrum features in a 3-D subspace shown in Figure 9. The features of a single user form a cluster, which clearly differs from other clusters that represent different users. The PCA subspace verifies the effectiveness of the spectrum-assisted method. Moreover, it only takes 19 ms to check the spectrum similarity constraints.

4.4 System Robustness

We conduct extensive experiments to study the system robustness, which may be impacted by vehicle driving states, the

# of Speakers	Idling	Local	Highway
1	100%	83.3%	58.3%
2	66.7%	58.3%	66.7%
3	75%	66.7%	75%
4	100%	100%	100%
Total	83.3%	73.8%	71.4%

Table 1: The Detection Accuracy for Different Number of Speakers Under Different Driving States.

placement of microphones, and driver’s sitting positions.

Vehicle Driving States. Three types of driving states are tested in our experiments: *idling*, *driving on local streets*, and *driving on highway*. Idling refers to running a vehicle’s engine but the vehicle is not in motion. The car is in a low-noise environment during idling. Driving on local streets means that the car runs at a low speed of around 20 miles per hour, where the car is usually in a medium-noise environment. Driving on highway indicates that the car runs at about 50 miles per hour on highway, with the highest level of environmental noise.

First, when detecting multiple speakers under each driving conditions, 42 voice segments are collected as the inputs of the autocorrelation algorithm to judge if the sample comes from multiple speakers. The experimental results for multiple speakers detection are shown in Table 1. We can see that the detection accuracy decreases gradually from the idling condition to the highway condition. Among them, the most significant change is the single speaker detection accuracy, which decreases considerably with different conditions. When driving on highway, the outside noise is so complex and unpredictable that the received signals contain a lot of noise peaks, which generate spurious high values in the autocorrelation calculation. Therefore, some signals from a single speaker may be incorrectly classified as multiple-source signals. The problem can be solved by using sound absorption material, better denoising algorithm, or multiple microphones scheme.

Second, we evaluate the detection accuracy of human voice under three driving states. Table 2 shows the driving states have little impact on the human voice detection, since the signal power is much greater than the noise power. Compared with the idling state, the detection accuracy only decreases by 3.26% when driving on the highway. Also, we discover that the driving noise has a higher influence on the time-domain verification than the frequency-domain verification, because the driving noise mainly affects the waveform in the time domain, not the statistical values in the frequency domain.

Third, one big challenge for the driver’s voice identification is the interference from outside noise. When the received signals are mixed with strong noise, there will be unexpected fluctuations in the cross-correlation function. These fluctuations will eventually offset the expected peaks, usually in the 0-offset direction due to the common-mode interference [43]. Table 3 shows the results of distinguishing the driver and the passengers. In the case of high interference, the driver’s voice

Driving State	Accuracy
Idling	97.46%
Driving on Local Street	96.75%
Driving on Highway	94.20%

Table 2: The Detection Accuracy of Human Voice under Different Driving States.

Voice Source		Idling	Local	Highway
Driver	Mean	-0.11	0.38	1.09
	Stdev	4.15	3.03	2.11
Front Passenger	Mean	-11.31	-10.99	-8.88
	Stdev	5.98	4.67	4.75
Rear Right Passenger	Mean	-8.02	-6.57	-5.31
	Stdev	4.04	3.29	5.00
Rear Left Passenger	Mean	-5.36	-5.30	-4.57
	Stdev	3.58	3.27	3.75

Table 3: The Peak Offsets for the Driver and Passengers under Different Driving States.

can still be distinguished from the passengers’ voice but the offset discrimination becomes moderate.

Relative Distance and Height from Microphones. In our scheme, the TDOA model is based on a far-field model. A quantitative experiment is conducted to evaluate the impacts of the distance between the voice source and the microphones and the height of microphones relative to the horizontal plane.

To evaluate the impacts of the voice source distance, 25 experiments are conducted and 2728 sample segments are acquired. In our experiments, 5 propagation angles (0 , $\pi/4$, $\pi/2$, $3\pi/4$, π) are tested. The testing range for voice source distance is from 1 foot to 5 feet with a spacing of 1 foot. From the experimental results illustrated in Figure 10(a), we can see that the measurement error is less than 2 offset units when the voice source distance is greater than or equal to 3 feet. Also, when the distance is less than or equal to 2 feet, the measurement error increases since the assumption of the far-field model is not applicable. However, inside vehicles, most sound sources are more than 2 feet away from the microphones.

To explore the effect of the voice source relative height on measurement accuracy, we conduct 20 experiments and collect 1795 voice segments. We evaluate the measurement error in 5 different propagation angles (0 , $\pi/4$, $\pi/2$, $3\pi/4$, π). Because of the far-field model, the horizontal distance between the microphones and the testing voice source is set to 3 feet, and the relative vertical height of the testing voice source is set to 0.5, 1, 1.5, and 2 feet, respectively. The experimental results are shown in Figure 10(b). With the propagation angle of $\pi/2$, the voice source relative height has little impact on the measuring accuracy, since the relative height does not introduce an additional distance difference. However, when the propagation angle is not $\pi/2$, the measuring error increases

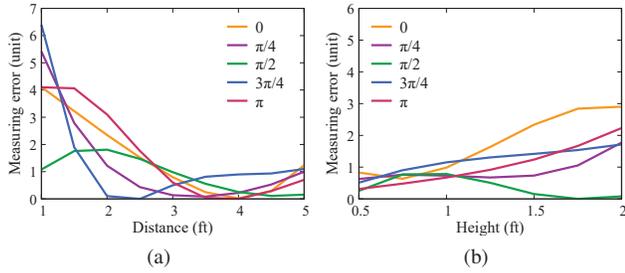


Figure 10: Measurement Accuracy on Distance and Height. (a) measurement error vs. relative distance; (b) measurement error vs. relative height.

with the relative height of the voice source because of the extra distance differences of arrival. The measuring error is more obvious when the propagation angle is close to 0 or π . However, the measuring errors in all cases are less than 3 offset units when the relative height of the voice source is not larger than 2 feet. Therefore, the relative vertical height of the voice source has minimal impact on our measurement.

Driver’s Sitting Positions. As the driver’s seat can be adjusted according to the driver’s preferences, the driver’s voice source would move forward or backward. In addition, according to different drivers’ driving habits, the driver’s position may lean towards left or right side. Therefore, we need to explore the influence of the different sitting positions on the measurement accuracy. As shown in Figure 11(a), the five most common sitting positions (normal, front, rear, left, right) are used to test the measurement accuracy. The microphones are positioned at a 45 degree angle to the front right of the driver’s normal position. For each sitting position, 4 sample instances are tested, and the experimental results are shown in Figure 11(b). In ideal circumstances, a voice signal comes from a normal position will have a zero offset value in the cross-correlation function. However, in the real-world situation, the measurement error is 1 offset value for the voice coming from the normal position. When the driver moves forward or left, the voice propagation angle will be slightly less than $\pi/4$, and the voice cross-correlation will have a positive peak offset. When the driver moves back or right, the voice propagation angle will be a little greater than $\pi/4$, and the voice cross-correlation will have a negative peak offset. The offset values of the front position are larger than those of other positions, while the offset values in the right position are smaller than others. As the absolute offset values in most cases are less than 3, different sitting positions have little impact on the detection results. Thus, the decision threshold s_T is set to be 2, which can successfully distinguish the driver’s voice even at different sitting positions.

4.5 Performance Overhead

Table 4 shows the the system performance overhead including running time and memory size for each detection step.

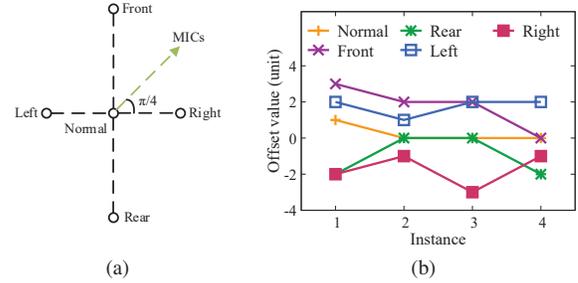


Figure 11: Experimental Results of Measurement Accuracy on Different Sitting Positions. (a) relative position of different sitting positions. (b) offset values of the instances with different sitting positions.

Detection Step	Running Time	Memory
Multi-speaker Detection	134 ms	111 MB
Human Voice Detection	47 ms	10 MB
Driver’s Voice Identification	33 ms	23 MB
Total Overhead Costs	214 ms	144 MB

Table 4: Performance Overhead for Detection Step.

Running time is measured as the average processing time for a single voice sample, and memory overhead measures the memory space occupied by the running program. The total running time for a voice sample is 214 ms in single-core mode with a 2.8GHz CPU, and the total occupied memory size is 144 MB. Since the running time is measured using slow Matlab code, we believe the optimized C code or assembly code may further reduce the running time.

Our system can be well supported by the modern in-vehicle computing platforms. For example, the in-vehicle embedded computing devices by Neusys Technology are configured with 2.3GHz-3.6GHz processor, up to 64GB DDR4 RAM [44]. FPGA-based or GPU-based hybrid Electronic Control Unit (ECU) can achieve hardware acceleration for in-vehicle computing [45] [46]. Since the driverless vehicle can deploy multiple ECU modules for different tasks [47], it is feasible to embed our system into a dedicated ECU module to avoid interfering other tasks running on other ECUs.

5 Discussions

Though our system design is customized to one popular sedan internal structure, it can be extended to other vehicle models or future driverless car models. In the left-driving countries (e.g., U.K. and Japan) where the driver’s seat is on the right, it is easy to adopt our system in those cars by mirroring the placement of the microphones and adjusting the location algorithm accordingly. Though most cars have four built-in speakers installed at four window corners, some models of cars have different numbers of speakers installed at different locations [48]. Since our speaker detection mechanism is effective on detecting the voice coming from more than one

speaker, it works well on different number of speakers.

Our current system design uses as few as two microphones close to each other. Thus, it is convenient to be installed as either built-in or external car microphone system with minor change on vehicle's interior design. The microphones in our prototype can divide the angle change between 0 and π into 28 regions; however, the microphones with a higher sampling rate may provide a fine-grained angle measurement. Also, a high-end microphone can reduce the noise in the background by supporting advanced denoising algorithms such as Fourier Bessel expansion [49]. It is also plausible to deploy more microphones (or a microphone array) in the future car designs. Thus, we can achieve a more accurate localization of the sound source in the three-dimensional space [50].

Our system integrates several detection methods that can be generalized and applied in other applications. For instance, the multiple speaker detection technique may be adopted in smart home systems to prevent malicious voice commands from household speakers. In the circumstances where sound sources have relatively fixed locations, our single voice source identification solution may be useful to determine the identity of the source. Moreover, our replay voice detection solution can be applied to enhance the security of voice-activated smart doors or other IoT devices.

6 Related Work

Automatic Speech Recognition (ASR) Systems. An automatic speech recognition system converts the speech signal into recognized words, which could be the inputs to natural language processing. Since it requires little special training and leaves hands and eyes free, ASR is ideal for drivers to issue commands to the vehicle systems. According to the capabilities of ASR systems, an ASR system can support either isolated word or continuous speech, read speech or spontaneous speech, speaker-dependent or speaker-independent [51], small vocabulary or large vocabulary, finite-state or context-sensitive language model [52], and high or low SNR [53].

Attacks on ASR Systems. The ASR systems are vulnerable to several voice-based attacks. For existing speaker identification solutions [54, 55], it remains as a challenge to defeat armored impersonation attacks [23, 56] and replay attacks [17, 18, 26]. Speech synthesis attack [57, 58] is a relatively complex method to perform attacks by a text-to-speech conversion. With the development of adversarial learning, more sophisticated attacks have emerged. Dolphin Attack [10, 59] utilizes ultrasonic modulation to move voice commands to an undetectable frequency band. Psychoacoustic model can be leveraged to generate the adversarial voices below the human perception threshold [6]. Voice commands can also be injected into voice controlled devices by laser modulation [60]. Some malicious voice commands can be understood by ASR systems, but not by humans [61]. Thus, attackers can hide voice commands in noise-like signals and control the mobile

voice recognition systems [8]. CommanderSong [9] demonstrates a more practical attack that embeds voice commands into music songs without being noticed by human beings.

Attacks on NLP module. Threats may also come from the natural language processing module of the voice control systems. Zhang et al. focus on the intent classifier in NLP module, generating semantic inconsistency by specific interpretation [62]. Moreover, an attack called *Skill Squatting* utilizes systematic error to hijack voice commands on Amazon Alexa and route them to a malicious third-party application with a similarly pronounced name [63]. Mitev et al. use skill-based Man-in-the-Middle attack to hijack conversation between Alexa and victims [64]. Attackers can also leverage voice masquerading attack to steal users information by impersonating as a legal application and communicating with the users [65].

Sound Source Localization. The most popular way of sound source localization is to utilize the time delays of arrival (TDOA) in different sound receivers [66]. Particularly, the direction-of-arrival (DOA) can be measured on a pair of microphones [67]. DOA techniques can be used to verify voice commands for IoT devices [68]. DOA is also utilized to detect articulator dynamic within a short distance for securing the mobile ASR systems [69, 70]. The 3-D sound source can be determined by using an array of multiple microphones [71, 72]. With different installations of microphone array, such as planar array [73] or rectangular prism [39], we may use different location estimation algorithms. An advanced method uses blind source separation technique to locate multiple sound sources simultaneously [74]. Moreover, sound source localization can also be achieved by other methods such as Gaussian mixture models [75] or golden section searching [76].

7 Conclusion

In this paper, we propose a secure in-vehicle ASR system called SIEVE to defeat adversarial voice command attacks on voice-controlled vehicles. We utilize the physical attributes of voices to distinguish the driver's voice from other adversarial voices in three steps. First, multi-source signals are filtered out according to the diffusion of autocorrelation on linear prediction residuals. Second, voice attacks from non-human speakers are filtered out by cross-checking both the frequency domain and time domain. Third, the driver's voice is determined from its propagation direction with a dual microphone. We implement a system prototype and conduct experiments in real cars. The experimental results show our system can achieve a high detection accuracy in real-world situations.

Acknowledgments

This work is partially supported by the U.S. ARO grant W911NF-17-1-0447, U.S. ONR grant N00014-18-2893, and NSFC grant 61572278. Jiahao Cao and Qi Li are the corresponding authors of this paper.

References

- [1] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Julia Kindelsberger, Li Ding, Sean Seaman, Hillary Abraham, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. MIT autonomous vehicle technology study: Large-scale deep learning based analysis of driver behavior and interaction with automation. *CoRR*, abs/1711.06976, 2017.
- [2] Tesla Autopilot. Wikipedia, the free encyclopedia, 2018. [accessed 20-November-2018].
- [3] Waymo. Wikipedia, the free encyclopedia, 2018. [accessed 20-November-2018].
- [4] Carplay. Wikipedia, the free encyclopedia, 2019. <https://en.wikipedia.org/wiki/Carplay>, [accessed December 2019].
- [5] Automated driving system. Wikipedia, the free encyclopedia, 2019. https://en.wikipedia.org/wiki/Automated_driving_system, [accessed December 2019].
- [6] Lea Schönherr, Katharina Kohls, Steffen Zeiler, Thorsten Holz, and Dorothea Kolossa. Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding. *CoRR*, abs/1808.05665, 2018.
- [7] N. Carlini and D. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 1–7, May 2018.
- [8] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 513–530, Austin, TX, 2016. USENIX Association.
- [9] Xuejing Yuan, Yuxuan Chen, Yue Zhao, Yunhui Long, Xiaokang Liu, Kai Chen, Shengzhi Zhang, Heqing Huang, XiaoFeng Wang, and Carl A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 49–64, Baltimore, MD, 2018. USENIX Association.
- [10] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 103–117, 2017.
- [11] Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury. Inaudible voice commands: The long-range attack and defense. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 547–560, Renton, WA, 2018. USENIX Association.
- [12] Liwei Song and Prateek Mittal. Inaudible voice commands. *CoRR*, abs/1708.07238, 2017.
- [13] Yuan Gong and Christian Poellabauer. An overview of vulnerabilities of voice controlled systems. *CoRR*, abs/1803.09156, 2018.
- [14] V. L. L. Thing and J. Wu. Autonomous vehicle security: A taxonomy of attacks and defences. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 164–170, Dec 2016.
- [15] Wonkyum Lee, Kyu J. Han, and Ian Lane. Semi-supervised speaker adaptation for in-vehicle speech recognition with deep neural networks. In *Interspeech 2016*, pages 3843–3847, 2016.
- [16] Rosa González Hautamäki, Tomi Kinnunen, Ville Hautamäki, Timo Leino, and Anne-Maria Laukkanen. I-vectors meet imitators: on vulnerability of speaker verification systems against voice mimicry. In *INTER-SPEECH*, 2013.
- [17] Z. Wang, G. Wei, and Q. He. Channel pattern noise based playback attack detection algorithm for speaker recognition. In *2011 International Conference on Machine Learning and Cybernetics*, volume 4, pages 1708–1713, July 2011.
- [18] J. Villalba and E. Lleida. Preventing replay attacks on speaker verification systems. In *2011 Carnahan Conference on Security Technology*, pages 1–8, Oct 2011.
- [19] V. Hautamäki, T. Kinnunen, F. Sedlák, K. A. Lee, B. Ma, and H. Li. Sparse classifier fusion for speaker verification. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(8):1622–1631, Aug 2013.
- [20] Nicholas Evans, Tomi Kinnunen, Junichi Yamagishi, Zhizheng Wu, Federico Alegre, and Phillip De Leon. *Speaker Recognition Anti-spoofing*, pages 125–146. Springer London, London, 2014.
- [21] A. G. Adami, R. Mihaescu, D. A. Reynolds, and J. J. Godfrey. Modeling prosodic dynamics for speaker recognition. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03)*, volume 4, pages IV–788, April 2003.

- [22] L. Ferrer, N. Scheffer, and E. Shriberg. A comparison of approaches for modeling prosodic features in speaker recognition. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 4414–4417, March 2010.
- [23] Johnny Mariéthoz and Samy Bengio. Can a professional imitator fool a gmm-based speaker verification system? *Idiap-RR Idiap-RR-61-2005*, IDIAP, 2005.
- [24] David Gerhard. Pitch extraction and fundamental frequency: History and current techniques. Technical report, 2003.
- [25] Marcin Witkowski, Stanislaw Kacprzak, Piotr Zelasko, Konrad Kowalczyk, and Jakub Galka. Audio replay attack detection using high-frequency features. In *INTERSPEECH 2017*, 2017.
- [26] Tomi Kinnunen, Md Sahidullah, Héctor Delgado, Massimiliano Todisco, Nicholas Evans, Junichi Yamagishi, and Kong Aik Lee. The ASVspoof 2017 challenge: Assessing the limits of replay spoofing attack detection. In *INTERSPEECH 2017, Annual Conference of the International Speech Communication Association, August 20-24, 2017, Stockholm, Sweden, Stockholm, SWEDEN*, 08 2017.
- [27] Zhizheng Wu, Nicholas Evans, Tomi Kinnunen, Junichi Yamagishi, Federico Alegre, and Haizhou Li. Spoofing and countermeasures for speaker verification: A survey. *Speech Communication*, 66:130 – 153, 2015.
- [28] P. V. A. Kumar, L. Balakrishna, C. Prakash, and S. V. Gangashetty. Bessel features for estimating number of speakers from multispeaker speech signals. In *2011 18th International Conference on Systems, Signals and Image Processing*, pages 1–4, June 2011.
- [29] 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 *2010 IEEE Symposium on Security and Privacy*, pages 447–462, May 2010.
- [30] J. Makhoul. Linear prediction: A tutorial review. *Proceedings of the IEEE*, 63(4):561–580, April 1975.
- [31] V. S. Ramaiah and R. R. Rao. Multi-speaker activity detection using zero crossing rate. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 0023–0026, April 2016.
- [32] M. Z. Ikram. Double-talk detection in acoustic echo cancellers using zero-crossings rate. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1121–1125, April 2015.
- [33] F. Gustafsson and F. Gunnarsson. Positioning using time-difference of arrival measurements. In *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, volume 6, pages VI–553, April 2003.
- [34] K. Kwak and S. Kim. Sound source localization with the aid of excitation source information in home robot environments. *IEEE Transactions on Consumer Electronics*, 54(2):852–856, May 2008.
- [35] Voice frequency. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Voice_frequency, 2019. [accessed April 2019].
- [36] Jesús Villalba and Eduardo Lleida. Detecting replay attacks from far-field recordings on speaker verification systems. In Claus Vielhauer, Jana Dittmann, Andrzej Drygajlo, Niels Christian Juul, and Michael C. Fairhurst, editors, *Biometrics and ID Management*, pages 274–285, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [37] Timbre. Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Timbre>, 2019. [accessed April 2019].
- [38] Maxima and minima. Wikipedia, the free encyclopedia, 2019. https://en.wikipedia.org/wiki/Maxima_and_minima, [accessed April 2019].
- [39] J. M. Valin, F. Michaud, J. Rouat, and D. Letourneau. Robust sound source localization using a microphone array on a mobile robot. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 2, pages 1228–1233 vol.2, Oct 2003.
- [40] F. Li and R. J. Vaccaro. Performance degradation of doa estimators due to unknown noise fields. *IEEE Transactions on Signal Processing*, 40(3):686–690, March 1992.
- [41] MATLAB Function Reference. wavwrite function. <http://matlab.izmiran.ru/help/techdoc/ref/wavwrite.html>. Accessed September, 2019.
- [42] Wen Ge, Xu Hongzhe, Zheng Weibin, Zhong Weilu, and Fu Baiyang. Multi-kernel pca based high-dimensional images feature reduction. In *2011 International Conference on Electric Information and Control Engineering*, pages 5966–5969, April 2011.
- [43] Common-mode interference. Wikipedia, the free encyclopedia, 2018. https://en.wikipedia.org/wiki/Common-mode_interference, [accessed December 2018].

- [44] Neosys Technology. In *Vehicle Computing*, 2019. <https://www.neosys-tech.com/en/product/application/in-vehicle-computing>, [accessed April 2019].
- [45] Javier Perez Fernandez, Manuel Alcazar Vargas, Juan M. Velasco Garcia, Juan A. Cabrera Carrillo, and Juan J. Castillo Aguilar. Low-cost fpga-based electronic control unit for vehicle control systems. *Sensors*, 19(8), 2019.
- [46] Pinar Muyan-Ozcelik and Vladimir Glavtchev. GPU Computing in Tomorrow’s Automobiles. https://www.nvidia.com/content/nvision2008/tech_presentations/Automotive_Track/NVISION08-GPU_Computing_in_Tomorrows_Automobiles.pdf, 2019. [accessed April 2019].
- [47] Electronic Control Unit. Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Electronic_control_unit, 2019. [accessed September 2019].
- [48] Vehicle Audio. Wikipedia, the free encyclopedia, 2018. https://en.wikipedia.org/wiki/Vehicle_audio, [accessed December 2018].
- [49] V. V. Baskar, B. Abhishek, and E. Logashanmugam. Emd-fb based denoising algorithm for under water acoustic signal. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 106–111, July 2014.
- [50] L. Wang, J. D. Reiss, and A. Cavallaro. Over-determined source separation and localization using distributed microphones. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(9):1573–1588, Sep. 2016.
- [51] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1053–1067, San Diego, CA, 2014. USENIX Association.
- [52] Youssef Bassil and Paul Semaan. ASR context-sensitive error correction based on microsoft n-gram dataset. *CoRR*, abs/1203.5262, 2012.
- [53] Christophe Ris and Stephane Dupont. Assessing local noise level estimation methods: Application to noise robust asr. *Speech Communication*, 34(1):141 – 158, 2001. Noise Robust ASR.
- [54] N. N. An, N. Q. Thanh, and Y. Liu. Deep cnns with self-attention for speaker identification. *IEEE Access*, 7:85327–85337, 2019.
- [55] Y. Lukic, C. Vogt, O. Dürr, and T. Stadelmann. Speaker identification and clustering using convolutional neural networks. In *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, Sep. 2016.
- [56] Rosa González Hautamäki, Tomi Kinnunen, Ville Hautamäki, Timo Leino, and Anne-Maria Laukkanen. I-vectors meet imitators: on vulnerability of speaker verification systems against voice mimicry. In *INTER-SPEECH*, 2013.
- [57] Phillip L. De Leon, Bryan Stewart, and Junichi Yamagishi. Synthetic speech discrimination using pitch pattern statistics derived from image analysis. In *INTER-SPEECH*, 2012.
- [58] Z. Ali, M. Imran, and M. Alsulaiman. An automatic digital audio authentication/forensics system. *IEEE Access*, 5:2994–3007, 2017.
- [59] M. Zhou, Z. Qin, X. Lin, S. Hu, Q. Wang, and K. Ren. Hidden voice commands: Attacks and defenses on the vcs of autonomous driving cars. *IEEE Wireless Communications*, pages 1–6, 2019.
- [60] Takeshi Sugawara, Benjamin Cyr, Sara Rampazzi, Daniel Genkin, and Kevin Fu. Light commands: Laser-based audio injection on voice-controllable systems. 2019.
- [61] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine noodles: Exploiting the gap between human and machine speech recognition. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT’15, pages 16–16, Berkeley, CA, USA, 2015. USENIX Association.
- [62] Yangyong Zhang, Abner Mendoza, Guangliang Yang, Lei Xu, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications. In *Proceedings of the 2019 The Network and Distributed System Security Symposium (NDSS ’19)*. Internet Society, 2019.
- [63] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC’18, pages 33–47, Berkeley, CA, USA, 2018. USENIX Association.
- [64] Richard Mitev, Markus Miettinen, and Ahmad-Reza Sadeghi. Alexa lied to me: Skill-based man-in-the-middle attacks on virtual assistants. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS ’19, pages 465–478, New York, NY, USA, 2019. ACM.

- [65] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *IEEE SP 2019*, 2019.
- [66] J. Ianniello. Time delay estimation via cross-correlation in the presence of large estimation errors. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 30(6):998–1003, December 1982.
- [67] Yunmei Gong, Lizhi Li, and Xiaoqun Zhao. Time delays of arrival estimation for sound source location based on coherence method in correlated noise environments. In *2010 Second International Conference on Communication Systems, Networks and Applications*, volume 1, pages 375–378, June 2010.
- [68] Logan Blue, Hadi Abdullah, Luis Vargas, and Patrick Traynor. 2MA: Verifying voice commands via two microphone authentication. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 89–100, New York, NY, USA, 2018. ACM.
- [69] Linghan Zhang, Sheng Tan, and Jie Yang. Hearing your voice is not enough: An articulatory gesture based liveness detection for voice authentication. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 57–71, New York, NY, USA, 2017. ACM.
- [70] Linghan Zhang, Sheng Tan, Jie Yang, and Yingying Chen. Voicelive: A phoneme localization based liveness detection for voice authentication on smartphones. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1080–1091, New York, NY, USA, 2016. ACM.
- [71] U. Klein and Trinh Quoc Vo. Direction-of-arrival estimation using a microphone array with the multichannel cross-correlation method. In *2012 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 251–256, Dec 2012.
- [72] Jie Yang, Simon Sidhom, Gayathri Chandrasekaran, Tam Vu, Hongbo Liu, Nicolae Cekan, Yingying Chen, Marco Gruteser, and Richard P. Martin. Detecting driver phone use leveraging car speakers. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom '11, pages 97–108, New York, NY, USA, 2011. ACM.
- [73] D. Ying, J. Li, Y. Feng, and Y. Yan. Direction of arrival estimation based on weighted minimum mean square error. In *2013 IEEE China Summit and International Conference on Signal and Information Processing*, pages 318–321, July 2013.
- [74] H. Heli and H. R. Abutalebi. Localization of multiple simultaneous sound sources in reverberant conditions using blind source separation methods. In *2011 International Symposium on Artificial Intelligence and Signal Processing (AISP)*, pages 1–5, June 2011.
- [75] L. Sun and Q. Cheng. Indoor sound source localization and number estimation using infinite gaussian mixture models. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1189–1193, Nov 2014.
- [76] C. Jung, R. Liu, and K. Lian. A fast searching algorithm for real-time sound source localization. In *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 1413–1416, Sept 2017.

μ SBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability

Majid Salehi

imec-Distrinet, KU Leuven
majid.salehi@kuleuven.be

Danny Hughes

imec-Distrinet, KU Leuven
danny.hughes@kuleuven.be

Bruno Crispo

imec-Distrinet, KU Leuven
Trento University, Italy
bruno.crispo@unitn.it

Abstract

A large portion of the already deployed Internet of Things (IoT) devices are bare-metal. In a bare-metal device, the firmware executes directly on the hardware with no intermediary OS. While bare-metal devices increase efficiency and flexibility, they are also subject to memory corruption vulnerabilities that are regularly uncovered. Fuzzing is an effective and popular software testing method to discover vulnerabilities. The effectiveness of fuzzing approaches relies on the fact that memory corruption faults, by violating existing security mechanisms such as MMU, are observable, thus relatively easy to debug. Unfortunately, bare-metal devices lack such security mechanisms. Consequently, fuzzing approaches encounter silent memory corruptions with no visible effects making debugging extremely difficult. This paper tackles this problem by proposing μ SBS, a novel approach that, by statically instrumenting the binaries, makes memory corruptions observable. In contrast to prior work, μ SBS does not need to reverse engineer the firmware. The approach is practical as it does not require a modified compiler and can perform policy-based instrumentation of firmware without access to source code. Evaluation of μ SBS shows that it reduces security analyst effort, while discovering the same set of memory error types as prior work.

1 Introduction

Recent years have witnessed the proliferation of Internet of Things (IoT) devices into nearly every aspect of our lives. According to a recent Gartner report [5], the number of connected IoT devices is expected to exceed the total number of humans by 2020. A large portion of these devices are bare-metal with the firmware running directly on the hardware. This approach can deliver energy-efficiency, extensible connectivity, and adequate computing power. However, most of these firmwares are implemented in type-unsafe languages such as C, C++, or Objective-C, that are prone to memory corruption vulnerabilities such as buffer overflows. This creates a very large attack surface in the IoT ecosystem.

Given the limited resources of bare-metal devices, traditional mitigation mechanisms for memory corruption vulnerabilities such as Control Flow Integrity (CFI) [24, 44], Address Space Layout Randomization (ASLR), and security policy reinforcement [57] are typically infeasible [16]. More importantly, many IoT devices today work in a real-time environment and must remain responsive to external stimuli (e.g., a health-care system, or a safety system in a car). These systems cannot accommodate the high run-time overhead incurred by most mitigation mechanisms. This highlights the importance of performing a vulnerability discovery process before the firmware is released, thus at testing time.

Fuzz-testing or Fuzzing is a testing solution for finding bugs and vulnerabilities. Fuzzing methods execute the application with randomly generated inputs and wait for vulnerability-exposing behaviors such as crashing or hanging. This behavior is the visible consequences of faulty states triggered by deployed security mechanisms such as Memory Management Unit (MMU). Unlike general purpose computers, bare-metal devices often lack such mechanisms due to their cost sensitivity and resource constraints. Accordingly, fuzzing bare-metal devices is extremely challenging to debug since memory corruptions may trigger no observable behaviors and thus cannot be discovered through fuzzing.

To address this problem, Muench et al. [42] integrated the black-box fuzzer Boofuzz [2] with a set of heuristics to recognize faults due to memory corruptions¹, but experimental results show it has still false positives and false negatives. Even more important, the proposed heuristics rely on information extracted from applying reverse engineering techniques and additional annotations provided manually by the analyst, all activities that are challenging and time consuming. Furthermore, they need to be applied for each new firmware under analysis.

Sanitizers [51], can be combined with fuzzing methods in order to make faulty states observable. Sanitizers instrument applications with memory check instructions to monitor all

¹In the interests of brevity we refer to a faulty state caused by memory corruption simply as a fault for the rest of the paper.

reads and writes during application execution. There are sanitizers that operate at the source code or compilation level, such as AddressSanitizer [46], while others, as Valgrind's Memcheck [48] that operate on machine code. Considering that the majority of bare-metal firmware are not open-source, source-based sanitizers are not the best choice in our context. Binary sanitization could be done either statically or dynamically. Dynamic binary sanitizers allow instrumentation of an application at runtime. However, such techniques are not widely deployable on the bare-metal devices mainly due to the high performance penalties and special software/hardware requirements. On the other hand, static binary sanitizers introduces lower overhead by instrumenting application binary statically. Unfortunately, at the time of writing none of the current binary sanitizers provides support for bare-metal devices.

This paper presents μ SBS, a novel approach that, by statically instrumenting bare-metal firmware binaries, makes memory corruptions observable. μ SBS provides a static binary instrumentation method and uses it for instrumenting memory instructions (i.e., sanitization). μ SBS allows to embed a given memory safety policy and to monitor all memory accesses, triggering observable warnings when a violation to the policy occurs. In summary, the paper makes following contributions:

- We present μ SBS, the first static binary sanitizer for bare-metal firmware. It avoids the complex and tedious work of reverse engineering firmware binaries.
- Using μ SBS, we make memory corruption faults observable also on bare-metal devices, thus facilitating their debugging.
- We developed a fully functional prototype of μ SBS for the ARM architecture which is the most widely used architecture in IoT devices. To foster further research, we make our μ SBS prototype available open source.
- We evaluated the effectiveness of μ SBS in catching the same classes of memory faults of prior work. We assess the feasibility of μ SBS by instrumenting 11 real-world firmware binaries. Evaluation results show that μ SBS correctly instruments all of the firmware binaries with reasonable execution over-head and size expansion.

2 Background and Motivation

In this section, we present a brief overview of memory corruption vulnerabilities and fuzzing as an approach to discover them, and discuss some limitations related to the architecture of bare-metal devices that motivate the need to extend and refine faults observability on such architectures.

2.1 Memory Corruptions and Fuzzing

Low-level systems software such as firmware is typically written in the C or C++ languages due to their efficiency and capability to fully control the underlying hardware. In such programming languages, developers must ensure that every memory access is valid, that no situation leads to the de-referencing of invalid pointers. However, in practice, developers frequently fail to meet these responsibilities and cause memory bugs that can be exploited by an attacker to alter the application behavior or even taking full control over the software stack.

In testing the security of such application, security analysts hardly have access to the source code. Fuzzing is one of the most effective testing methodologies to find memory corruption vulnerabilities in Commercial Off-The-Shelf (COTS) applications. Fuzzing executes the application binary file with random inputs to look for unexpected application behavior such as crashes that are immediate consequences of faulty states. The ability of observing such crashes is the prerequisite for fuzzing to work. In general purpose computer systems, equipped with OS security mechanisms and hardware features such as stack canaries, Data Execution Prevention (DEP), Memory Management Unit (MMU), and Memory Protection Unit (MPU), memory violations trigger a crash upon a fault. Possible ways to observe such crashes are: (1) Observing exit status: the execution of the device or application under test is terminated and an error message is generated for tracing. (2) Catching the crashing exception: the crashing signal can be caught by overwriting an exception handler. (3) Leveraging mechanisms provided by the OS: the OS-level debugging interfaces such as ptrace can be used in order to observe application execution and detect crashes.

2.2 Bare-metal Embedded Devices

Among different classes of embedded devices, bare-metal devices are designed for low cost and low power operation. Such devices are deployed in many application areas ranging from automotive and industrial control systems to medical devices. Bare-metal devices execute a single statically linked binary firmware providing a specific application logic as well as system functionality without the use of an underlying abstraction such as an operating systems. However, it is challenging for bare-metal devices to support security properties in practice, due to limited energy, memory and computing resources. For example, this class of devices rarely provide a Memory Management Unit (MMU) and firmware modules have access to the entire shared memory space in a privileged mode. Therefore, compromising one firmware module gives an attacker arbitrary read/write access to the whole system with no observable side-effects. Unrestricted read/write primitive enables the attacker to redirect the control-flow of firmware or directly overwrite sensitive data.

Table 1: Hardware protection mechanisms supported by representative core families.

Core Family		Hardware Protection Mechanism		
		MPU	MMU	DEP
ARM	ARM 1 to ARM 7	✗	✗	✗
	ARM 7EJ	✗	✗	✗
	ARM Cortex R	✓	✗	✓
	ARM Cortex M	~	✗	✓
PIC	PIC 10 to PIC 24	✗	✗	✗
	dsPIC	✗	✗	✗
AVR	ATiny	✗	✗	✗
	ATmega	✗	✗	✗
	ATxmega	✗	✗	✗
8051	Intel MCS-51	✗	✗	✗
	Infineon XC88X-I	✗	✗	✗
	Infineon XC88X-A	✗	✗	✗
MSP430	MSP430x1xx to MSP430x6xx	✗	✗	✗
	MSP430FRxx	✓	✗	✗

✓: it is supported by all microcontrollers in the given family.
 ~: it is supported by some microcontrollers in the given family.
 ✗: it is not supported by any of them.

As a more concrete investigation of the hardware security feature support (i.e., MMU, MPU, and DEP), we conducted an analysis of 29 SoC core families. Our selection aims to provide a representative sample of major architectures and vendors in the embedded space across industry verticals including unmanned aerial vehicle (UAV), unmanned ground vehicle (UGV), remotely operated underwater vehicle (ROV), real-time 3D printer controllers and real-time Internet of Things (IoT) devices.

According to our analysis, none of the SoCs is designed to employ MMU. A number of SoCs optionally provide basic memory protections using MPU. However, even with the existence of MPU, configuring it from the application is not a straightforward task, leading the developers to ignore using this functionality. Table 1 summarizes the results of our analysis by mapping out core families architectural style and hardware security functionalities.

2.3 Fault Observability in Bare-metal Devices

Contemporary general purpose computers have plenty of mechanisms that makes faulty states observable (e.g., segmentation faults caused by an MMU). Most bare-metal devices, instead, do not have such mechanisms due to their limited I/O capabilities and architecture. In fact, most memory corruption events are *silent* and do not lead to an immediate crash of the firmware or any observable event. Thus the firmware can continue the execution with no visible effect or it will lead eventually to a crash (i.e., I/O error) that is however very difficult to debug. It is challenging to infer if the crash was due to an early memory violation or to an I/O error.

Motivating Example. To better understand the problem, we use a popular bare-metal firmware, Broadcom Wi-Fi SoC as a motivating example. This firmware is present in both mobile devices and Wi-Fi routers for handling the lower layers of Wi-Fi and Bluetooth protocols. The Broadcom Wi-Fi SoC executes on ARM Cortex-R processor. As reported by Google Project Zero [6] in CVE-2017-0561 [4], the firmware has a remote code execution vulnerability that enables a remote attacker to execute arbitrary code and escalate to control over the entire system. In the code snippet shown in Listing 1, SoC firmware performs a *memcpy* into the allocated memory object *buffer*, using the *ft_ie* length field. Since the *ft_ie* length field is not verified prior to the copy, this allows an attacker to exceed the *buffer* and trigger a buffer overflow.

```

uint8_t* buffer = malloc(256);
...
uint8_t* linkid_ie = bcm_parse_tlvs(..., 101);
memcpy(buffer, linkid_ie, 0x14);
...
uint8_t* ft_ie = bcm_parse_tlvs(..., 55);
memcpy(buffer + 0x18, ft_ie, ft_ie[1] + 2);

```

Listing 1: A remote code execution vulnerability in the Broadcom Wi-Fi firmware could enable a remote attacker to execute arbitrary code within the context of the Wi-Fi SoC.

As a proof of concept we exploited this vulnerability similarly to [6] and sent malicious inputs that triggered the buffer overflow vulnerability. Nonetheless, the firmware did not crash and continued to function normally with no observable side-effects. This is mainly due to the fact that Broadcom Wi-Fi SoC lacks all basic memory protection mechanisms

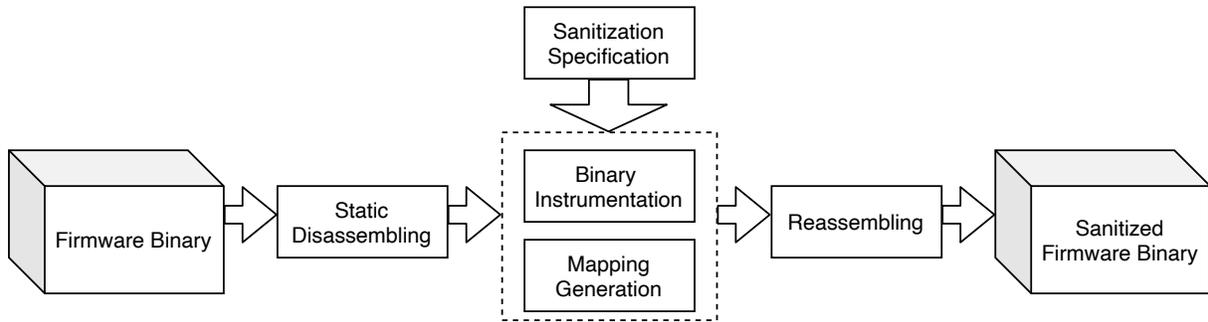


Figure 1: Workflow of μ SBS.

including access permission protection by means of either MMU, MPU or more advance protections like stack canaries.

There exist several approaches to make faulty states observable. Sanitizers [51] monitor the actual execution of an application in order to observe faulty states as they happen. They enforce *spatial* memory safety by detecting dereferences of pointers that do not target their intended referent, or enforce *temporal* memory safety by detecting dereferences of pointers that target a referent that is no longer valid. Sanitizers implement memory safety policies by embedding inlined reference monitors (IRM) into the application through instrumentation. IRMs mediate and monitor every memory accesses and memory object (de)allocations instructions. IRMs can be embedded at either source code or binary level. Source-based sanitizers [30, 33, 46, 53, 60] are not widely deployable on bare-metal devices due to the unavailability of their firmware source code since they are often proprietary.

Binary images of bare-metal firmware are often available to the analyst since they can be acquired by directly extracting from the physical device using debugging port (e.g., JTAG interface) or downloading and unpacking update packages available on many vendors websites. Thus, binary sanitization [23, 34, 48] is the only viable option for enforcing memory safety in bare-metal firmware. Dynamic binary sanitizers read application code, instrument it, and translate it to machine code while the application executes. However, these approaches do not generate a standalone instrumented binary and sanitization process has to be done again each time the application executes. In addition, they have significant run-time and space overhead which is a critical problem for bare-metal devices which have limited processing power and a small memory space. This overhead can essentially be attributed to the dynamic translation process. This issue can be addressed by instrumenting applications statically using a binary sanitizer, which we believe is a promising solution for our requirements.

Until now, there is no binary sanitizer for bare-metal firmware. Based on this limitation of previous work, we propose a novel and tailored automated method for making faults observable in bare-metal devices.

3 Static Binary Sanitization for Bare-metal Devices

Figure 1 illustrates a high-level overview of our approach, with the different components and their interactions. There are three main phases: the *static disassembling*, the *binary instrumentation*, and the *reassembling*.

The first step of our μ SBS workflow is *static disassembling* (§ 3.1). It disassembles the raw binary and decodes instructions using a linear disassembly method. The second phase is *binary instrumentation* (§ 3.2) that instruments the firmware binary based on the sanitization specification. Sanitization specification (§ 3.3) contains instrumentation information determining what instructions will be inserted or replaced in order to embed IRMs and monitor every memory access. In other words, μ SBS statically instruments every memory access with a runtime check to verify if it is an access to an allowed address. If not, our fault handler raises a warning close to the location of the bug to guide the follow-up security analysis to find out the root cause of the fault. The last step is the *reassembling*, that takes the instrumented assembly code and reassembles it as a working binary using off-the-shelf assemblers. In the following sections, we describe the most important and challenging aspects of μ SBS design.

3.1 Static Disassembling

Disassembly is referred as the process of parsing executable region of binary file from beginning to the end and decoding all encountered bytes into their raw textual representation. There are two popular types of disassembly approaches: linear sweep and recursive traversal disassemblers [19, 37, 45]. Linear sweep decodes all encountered bytes as instructions by sweeping the entire code section. Recursive traversal disassembles instructions following control flow transfers (e.g., jumps and calls). It is challenging to correctly and completely disassemble arbitrary code. This is mainly due to the fact that, in hand-written assembly and modern compilers, code and data can be interleaved and there is no syntactic distinction whereby the disassembler may distinguish them. However,

```

#include<stdio.h>
void printer_ver1 (float var) {
    printf("%.10f\n", var);
}
void printer_ver2 (float var) {
    printf("%.50f\n", var);
}
void (*printers[2])(float) = {printer_ver1,
    printer_ver2};

int main (void)
{
    static float var = 4e-34;
    static int (**ptr) (float)= &printers;
    (*(ptr+1))(var);
    return 0;
}

```

(a) Source code of running example.

```

0x804816c <printer_ver2>:
...
0x8048198:    bl    8048688 <printf>

0x80481b0 <main>:
...
0x80481bc:    movw r3, #64668 ; 0xfc9c
0x80481c0:    movt r3, #1
0x80481c4:    ldr  r3, [r3]
0x80481c8:    add  r3, r3, #4
0x80481cc:    ldr  r2, [r3]
0x80481d0:    movw r3, #64672 ; 0xfca0
0x80481d4:    movt r3, #1
0x80481d8:    ldr  r3, [r3]
0x80481dc:    mov  r0, r3
0x80481e0:    blx  r2

```

(b) Partial assembly code of running example.

```

Hex dump of section .data:
0x0001fc90 00000000 28810408 6c810408 94fc0100
0x0001fca0 3dec0408 00000000 00000000 94ff0100

```

(c) Hexdump of .data section.

Figure 2: A running example that covers three main challenges of binary instrumentation and illustrates four general classes of references (C2C, C2D, D2D, D2C) in assembly code.

Andriess et al. [18] noted that such cases are exceedingly rare and disassemblers achieve close to 100% accuracy for instruction disassembly from compiler-generated binaries. Therefore, we applied a linear sweep disassembly algorithm to our evaluation set.

3.2 Binary Instrumentation

The μ SBS *binary instrumentation* component takes the disassembled file and sanitization specification as inputs with the aim of statically inserting a number of memory check instructions to catch memory corruption vulnerabilities. However, binary instrumentation introduces challenges that are not present when modifying source code. Specifically, when instructions are inserted or removed at the source code level, the compiler will redo the linking process to rearrange code and data in memory. In binaries, inserting or removing instructions causes addresses to change and breaks the binary file due to the lack of linkage information. The symbol and relocation information, that is used in the linking process to ensure that application elements can correctly refer to each other, are discarded by the compiler once finished. In the following, we provide the details of practical challenges in designing μ SBS *binary instrumentation* component and our solutions tackling them.

The core process of instrumenting binaries is the ability to relocate any binary code without any relocation and metadata information. There are three main challenges in reloca-

tion procedure to avoid breaking the binary file. To describe these challenges clearly, Figure 2 shows a running application alongside its disassembly and the hex dump of its data section. This application declares a float variable *var* with an initial value of 4e-34 and prints that with two different formats (i.e., *printer_ver1* and *printer_ver2*). The first challenge is recognizing static addresses. There is no syntactic distinction to disambiguate between reference and scalar type for immediate values and updating references to the new targeted addresses. In our application, the compiler stores 3dec0408 in data section as the binary representation of *var*. Since our application has a code section with memory addresses ranging from 0x8000000 to 0x8100000, this immediate value can be considered as a pointer (0x804ec3d) on little-endian machines; this will irremediably corrupt the image if we update it after binary instrumentation.

The second challenge is relocating static addresses after instrumentation. For example, as illustrated in our application disassembly, the *printer_ver2* function calls the *printf* function with static address 0x8048688. It is clear that the insertion of instructions into, or removal of instructions from disassembly code can break this static address. The third challenge is determining dynamically referenced memory addresses. Contrary to static memory addresses that are explicit, the target addresses of some references are computed dynamically at runtime and they can not be updated statically. As shown in our application disassembly, the reference target *r2* is computed dynamically.

To tackle these challenges, we categorize all references into four general classes: Code-to-Code (C2C), Code-to-Data (C2D), Data-to-Code (D2C), and Data-to-Data (D2D). In our application, the `ldr r3, [r3]` instruction accesses to the address `0x0001fc9c` in order to retrieve the `printers` array address from data section (C2D). `94fc0100` hex value at address `0x0001fc9c` is a pointer to the `printers` array in data section (D2D). The second element (`6c810408`) of the `printers` array points to the `printer_ver2` function in code section (D2C), which calls the `printf` function at address `0x8048688` (C2C).

C2D and D2D references. Due to the fact that there is no need to perform instrumentation on the original data space, we can preserve the starting addresses of data sections intact. By doing so, we may easily ignore and handle C2D and D2D references.

C2C and D2C references. Since insertion of instructions causes stretched code space, μ SBS adds a new expanded code section (`.newsec`) at a new entry point. As demonstrated in Algorithm 1, μ SBS iterates all disassembled instructions and rewrites them intact in the `.newsec` section. Also, in the meanwhile, μ SBS performs instrumentation and inserts new memory check instructions in the `.newsec` section.

μ SBS adjusts all branch instructions target addresses while rewriting them in the `.newsec` section. Each direct branch instruction with an immediate operand can easily point to the new address by changing its offset statically. However, indirect branch instructions have multiple possible target addresses and therefore needs some sort of target-prediction mechanism. Unlike many prior efforts [36, 54, 55], we observe that while it is challenging to statically identify targets of indirect branch instructions, we can instead perform a dynamic lookup at runtime. It is mainly due to the fact that the precise target addresses are known at runtime. Specifically, we provide a mapping table from the old code section to `.newsec`. By doing so, we can modify each indirect branch instruction to search for the new target address in the mapping table after the old target address has been computed at runtime.

To generate the mapping table, it is first required to know each instruction size that is present in the `.newsec` section. Therefore, we record any changes to instructions and sizes while rewriting them in the `.newsec` section. More specifically, we generate a mapping table from each address in the old code section to the size of the corresponding rewritten bytes in the `.newsec` section. Afterwards, we are able to adjust reference targets by converting each size record in the mapping table to the corresponding offset in the `.newsec` section. Essentially, we add a level of indirection by replacing all indirect branch instructions with a direct branch to the mapping routine and consulting the mapping table for computing the new target address.

For instance, in Figure 2, the runtime value of indirect branch (`blx r2`) target address is `0x804816c`. As shown in Figure 3, μ SBS rewrites instructions in the `.newsec` section with base address `0x8200000`. At runtime, the mapping rou-

Algorithm 1: Generating a new code section

```

newsecGenerator (Insts)
  inputs : Insts = inst1 ... instn
  output : newsec section
  foreach disassembled instruction insti ∈ Insts do
    if insti.type is a branch_instruction then
      if insti.ref is static then
        insti.ref := AdjustTarget(insti.ref);
        WriteInst(newsec, insti)
      else
        WriteInst(newsec, mapping_instructions)
      else if SanitizationSpecification(insti) then
        if insti.type is a memory_allocation then
          WriteInst(newsec, redzone);
          WriteInst(newsec, insti);
          WriteInst(newsec, redzone);
        else
          WriteInst(newsec, metadata_check);
          WriteInst(newsec, insti);
      else
        WriteInst(newsec, insti)
  return newsec;

```

tine looks for `0x804816c` entry in mapping table in order to find the offset of new target address (`0x81d0`), and then returns new translated target address (`0x8200000 + 0x81d0 = 0x82081d0`). Finally, firmware jumps (`ldr pc, [sp, #-4]`) to the translated address.

3.3 Sanitization Specification

The sanitization specification determines which exact instructions should be instrumented by μ SBS. AddressSanitizer [46] and Valgrind’s Memcheck [48] are the most widely adopted sanitizers for detecting memory safety violations in practice [52]. Inspired by these approaches, μ SBS utilizes a metadata store that keeps the status of allocated memory bytes. μ SBS surrounds every memory value with a so-called red-zone representing out-of-bounds memory and marks it as invalid memory in the metadata store. Then, μ SBS instruments every memory instruction (i.e., load and store) in order to consult the metadata store whenever the firmware attempts to access memory. Any access to a red-zone or to an unallocated memory region is considered as a memory corruption vulnerability and triggers a warning close to the location of the bug.

The current version of μ SBS sanitizer enables us to observe faulty states caused by various types of spatial and temporal memory corruptions including: (1) Overrunning and under-running heap blocks. (2) Overrunning the top of the stack. (3) Accessing memory after it has been freed. (4) Using memory values that have not been initialized or that have been derived

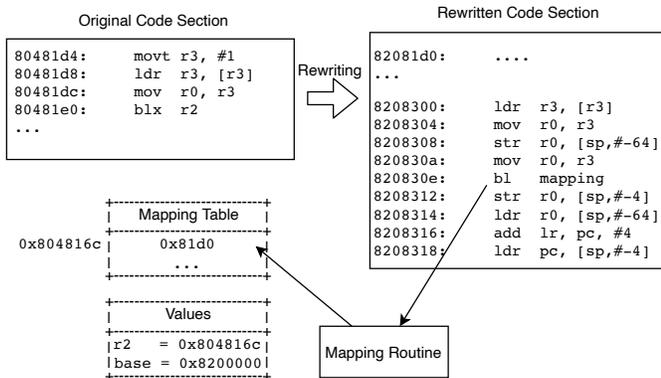


Figure 3: Mapping procedure: all indirect branch instructions (e.g., `blx r2`) are firstly redirected to the mapping routine which looks for new offset (0x81d0) corresponding to the old target address (0x804816c) in mapping table.

from other uninitialized values. (5) Incorrect freeing of heap memory, such as double-freeing heap blocks.

4 Implementation

We have implemented a proof-of-concept of μ SBS for the ARMv7-M architecture [1], which covers a large share of microcontrollers (i.e., Cortex-M3/4/7) for embedded platforms [35]. The following outlines the technical details of the implementation based on the design described in the previous section.

4.1 Binary Instrumentation

We implemented μ SBS *binary instrumentation* component with a total of 1609 LOC in Python language using the Capstone framework [3] as our underlying linear disassembler engine. We used the `pyelftools` [8] and `pwntools` [7] open source frameworks for parsing the ELF files and reassembling instrumented assembly code respectively.

As mentioned in § 3.2, μ SBS generates a new stretched code section (*.newsec*) and executes firmware from its new entry point. μ SBS rewrites all instructions of the old code section together with new inserted instructions in *.newsec*. However, inserting new instructions may push a target address beyond the reach of the instruction referencing it. To resolve this issue, μ SBS replaces every referencing instruction by its substitute which allows for larger offsets. For instance, it replaces *b* branch instructions with *b.w* to generate a 32-bit instead of 16-bit instruction. Furthermore, it is necessary to rewrite all branch instructions to handle the challenges mentioned in § 3.2.

To safely handle reference targets and adjust the binary layout, μ SBS rewrites all direct branch instructions by statically changing their offsets. Thereafter, it rewrites all indirect

branch instructions to deploy the dynamic mapping procedure. To do so, every indirect jump instruction with target address *label* must be rewritten by instructions shown in Listing 2.

```

str r0, [sp, #-64]
mov r0, label
bl mapping
str r0, [sp, #-4]
ldr r0, [sp, #-64]
add lr, pc, #4
ldr pc, [sp, #-4]

```

Listing 2: Rewritten instructions for every indirect jump instruction with target address *label*.

Similarly, every indirect call instruction with target address *func* must be rewritten by instructions shown in Listing 3. To store the return address, μ SBS loads into Link Register (LR) the address of the instruction following the call to `[sp, #-4]`.

```

str r0, [sp, #-64]
mov r0, func
bl mapping
str r0, [sp, #-4]
ldr r0, [sp, #-64]
add lr, pc, #4
ldr pc, [sp, #-4]

```

Listing 3: Rewritten instructions for every indirect call instruction with target address *func*.

μ SBS translates every indirect branch into the *mov* and direct call instructions. The *mov* instruction puts the old target address into register *r0* and the direct call goes to the mapping routine which searches for the offset corresponding to the old target in the mapping table. If the search succeeds, it will jump to the translated target address in *.newsec*.

Finally, μ SBS uses LIEF framework [9] to add *.newsec* and mapping routine to the original firmware ELF file. The LIEF framework modifies the ELF header and creates a new code segment containing the *.newsec* code section.

4.2 Sanitization

We implemented the process of generating a sanitization specification with 687 LOC in the Python language. Once the firmware binary is disassembled, μ SBS extracts all object allocations and memory accesses (i.e., LDR and STR) and stores them in the sanitization specification file. The *binary instrumentation* component interprets the sanitization specification file for instrumenting all memory accesses with memory check instructions to consult the metadata store. The memory check instruction computes the address of the corresponding metadata byte, loads that byte and checks whether it is valid. For efficiency reasons, μ SBS deploys a similar metadata management mechanism to AddressSanitizer by storing 1 byte of metadata for every 8 bytes of firmware memory. In this case, the metadata mapping accords with Formula 1 where *meta_base* is the base address of the metadata store

Table 2: Comparison of fault observability between μ SBS and the heuristic-based method proposed by Muench et al. [42].

Memory Corruptions	Muench et al.	μ SBS
Null Pointer Dereference	✓	✓
Stack-based buffer overflow	✓	✓
Heap-based buffer overflow	✓	✓
Format String	✓	✓
Double Free	✓	✓

and `block_addr` is the address of the memory block.

$$\boxed{meta_addr = meta_base + (block_addr \gg 3)} \quad (1)$$

5 Evaluation

We evaluated μ SBS from three different angles: (1) Whether it can make faulty states observable on binary firmware in an automatic fashion. (2) Whether it can rewrite the code section of binary firmware without breaking its functionality. (3) Assessing the runtime performance and size expansion of rewritten binaries in practice.

To that end, we investigated μ SBS ability to catch the same class of memory bugs of the state-of-the-art [42] based on the WYCNINWYC vulnerable application [15, 42] (§ 5.1). We also conducted an evaluation on 11 real firmware images to check the correctness of the rewritten binaries by using the standard test suite that provided by the vendor of each firmware (§ 5.2). Finally, we measure the runtime performance of our rewritten and instrumented binaries (§ 5.3).

5.1 Effectiveness

We designed and performed this experiment to verify whether μ SBS can successfully make the same class of memory corruptions observable compared to state-of-the-art fault observation method proposed by Muench et al. [42].

Experiment Setup: We used WYCNINWYC application, developed and used by Muench et al. as a testbench, in order to obtain comparable results. The WYCNINWYC application is a vulnerable implementation of an XML parser containing five different instances of spatial and temporal memory corruptions. Experiments are performed on a same development board, STM32-Nucleo L152RE [14] featuring an ARM Cortex-M3 CPU.

Experiment Results: We instrumented the WYCNINWYC application using the μ SBS sanitizer and collected the statistics and results, including the observability of faulty states caused by memory corruptions. As shown in Table 2, μ SBS caught all faulty states without the need for reverse engineering or advanced data-flow analysis techniques as required by the method of Muench et al. [42].

5.2 Feasibility

We performed this experiment to verify the correctness of the μ SBS design and its application to large real-world bare-metal firmware. We rewrote the code sections of 11 binary firmware images without sanitization and observed whether all rewritten binaries executed correctly and produced identical output to the original.

Experiment Setup: We selected 11 real bare-metal binary firmware images for different applications, ranging from cameras to industrial control systems. These are full-fledged firmware and demonstrate the use of a diverse set of peripherals including an LCD Display, Microphone, Camera, Serial port, Ethernet and SD card. They collectively cover ARM Cortex-M3 and Cortex-M4 microcontrollers. In what follows, we provide a brief description of each firmware.

Audio-Playback firmware is developed for playing audio files by reading data from USB and sending it to the audio codec. *LCD-Display* is a firmware for reading and displaying a series of bitmaps from an SD card to the LCD. *LCD-Animate* displays animated pictures saved on a microSD card on the LCD. To create animated pictures, the firmware displays an images sequence with a determined frequency on the LCD. *Camera-USB* uses the Digital Camera Interface (DCMI) to connect with a camera module and display pictures on an LCD in continuous mode while also saving these pictures on the USB device. *FatFs-uSD* creates a FAT file system on the microSD and uses FatFs APIs to access the FAT volume in order to perform writing and reading of a text file. *TCP/UDP-Echo-Client/Server* are four firmware for running TCP/UDP echo client/server applications over Ethernet based on LwIP, a popular TCP/IP stack for embedded devices. *mbed-TLS* firmware runs an SSL client application based on the mbed-TLS crypto library and the LwIP TCP/IP stack for the STM32F4 family. PLC (Programmable Logic Controller) is a family of embedded devices for controlling critical processes in industrial environments. The *ST-PLC* firmware implements a PLC that executes uploaded ladder logic programs. The ladder logic program is uploaded to the microcontroller from an Android application via WiFi (ladder logic is a common PLC programming language).

All of these firmware images are provided with the development boards and written by STMicroelectronics [10]. Experiments are performed on STM32-Nucleo F401RE [11], STM32F479I-Eval [12], and STM32F4Discovery [13] development boards featuring an ARM Cortex-M4 CPU and STM32-Nucleo L152RE [14] featuring an ARM Cortex-M3 CPU.

Experiment Results: We executed both the rewritten version and the original version of the firmware on the test suit shipped with the firmware and compared their functional correctness. All of the rewritten firmware passed the functionality test and ran correctly, producing the same result as the original firmware.

Table 3: Statistical metrics of μ SBS binary rewriting when applying to bare-metal firmware.

Firmware	MCU	Dir. Inst.	Ind. Inst.	Code (KB)	Rew. Code (KB)	Size Inc. (%)
Audio-Playback	STM479I-Eval	1853	250	132	195	24
LCD_Display	STM479I-Eval	809	103	48	57	10
LCD_Animate	STM479I-Eval	803	103	48	56	10
FatFs_uSD	STM4Discovery	575	150	23	29	6
TCP_Echo_Client	STM479I-Eval	1407	132	79	91	14
TCP_Echo_Server	STM479I-Eval	1384	132	77	89	14
UDP_Echo_Client	STM479I-Eval	1341	132	76	88	14
UDP_Echo_Server	STM479I-Eval	1310	130	75	87	14
Camera-USB	STM479I-Eval	1003	163	70	81	13
mbed-TLS	STM401RE Nucleo	3218	338	171	215	20
ST-PLC	STM479I-Eval	2275	373	168	231	67

Table 3 presents the rewriting statistics and the modifications made by μ SBS to the binary firmware images. The column under *Dir. Inst.* in the table represents the count of direct branch instructions, including calls and jumps, that are statically rewritten by changing their offsets. The column under *Ind. Inst.* represents the count of indirect branch instructions redirected to the mapping routine by μ SBS.

Additionally, columns *Code* and *Rew. Code* represent the sizes of original code section and the rewritten code section (*.newsec*) respectively. The code section size overhead correlates positively with the number of indirect branch instructions due to the rewriting procedure that replaces each indirect branch with 6/7 instructions (§ 4.1). Furthermore, the fixed overhead of the mapping table and mapping routine play a significant role in the size overhead of the binary file (last column). In fact, the size overhead in percent will be less for large firmware compared to small images. For example, the original size of the *ST-PLC* firmware is 1.1MB and has 67% overhead, while *mbed-TLS* is 3.5MB and has only 20% overhead.

5.3 Performance

μ SBS with no sanitization slows down firmware execution for two reasons. First, the firmware is statically instrumented and there is an additional direct call added for each indirect branch instruction to call the mapping routine. Second, μ SBS dynamically searches for the new target address of every indirect branch instruction in the mapping table which incurs runtime overhead. In this section, we evaluate the execution overhead of rewritten binaries without sanitization. In addition, we also measure the execution overhead incurred by our sanitization procedure and the processing time of μ SBS itself.

Each firmware in our benchmark was instrumented and executed twenty times. For example, we executed *LCD_Display* original and instrumented firmware for displaying 5 images twenty times. Figure 4 presents the runtime overhead results. The average slowdown for our benchmark without sanitiza-

tion is 8.5%. The second bar in Figure 4 presents the execution overhead of instrumented firmware with the μ SBS sanitizer. It increases the execution overhead to an average of 32.5% compared to rewritten firmware without sanitization. *Audio-Playback*, *mbed-TLS*, and *ST-PLC* are memory-intensive firmware, and therefore we expected the high overhead results arising from a large number of memory accesses that are expensive after being instrumented and checked. While, these overheads are not negligible, we believe that they are reasonable as this overhead is only incurred on the devices under security test and not the devices that are actually deployed on the field.

Figure 5 presents how long it takes μ SBS to rewrite and sanitize firmware binaries. As expected, larger binaries take more time to be processed. On average, μ SBS spends 5.72 seconds on binaries in our benchmark. We regard this as a promising result compared to state-of-the-art fault observation method [42]. The efficiency of μ SBS makes it a practical tool for the large-scale sanitization of firmware binaries.

6 Related Work

This section provides an overview of the state-of-the-art. Related work can be categorised in works related to: (1) fault observation and (2) binary rewriting.

6.1 Fault Observation

Muench et al. [42] proposed the only existing method for fault observation in embedded systems by introducing six heuristics such as heap object tracking. They implemented these heuristics on top of a combination of the Avatar [61], PANDA [29], and Boofuzz [2] frameworks. However, these heuristics suffer from their reliance on a variety of information such as: memory accesses, memory mappings, executed instructions, register state and allocation and deallocation functions. This information must be extracted from target bi-

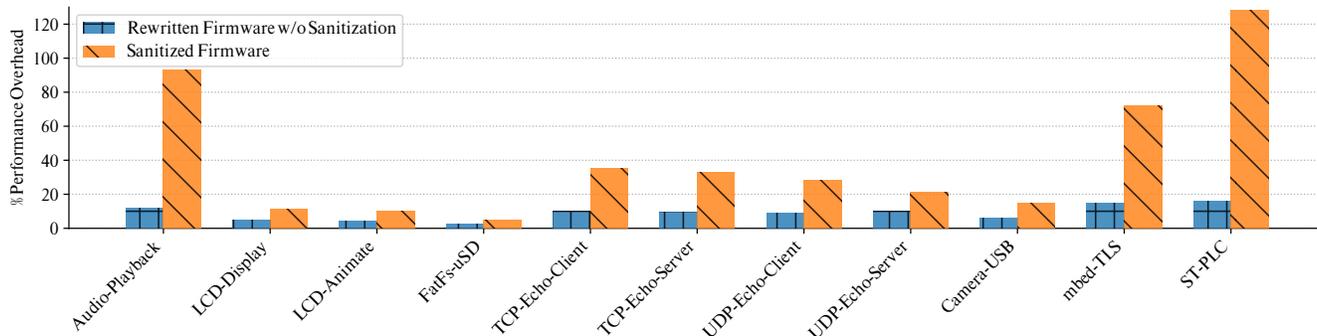


Figure 4: Performance impact of μ SBS on 11 real firmware binaries with and without sanitization.

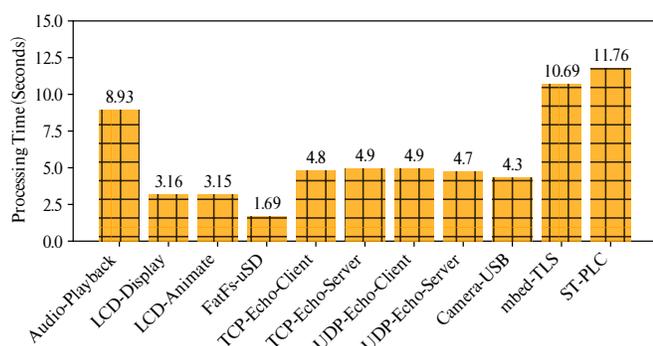


Figure 5: μ SBS processing time for 11 real firmware binaries.

nary through reverse engineering and advanced static analysis, which adds both imprecision and complexity. Furthermore, applying heuristics for fault observation results in false positives and false negatives. Instead, μ SBS approach turns out to be not only lightweight, avoiding heavyweight manual firmware analysis, but also reliable, being capable of providing platform-agnostic fault observation.

6.2 Binary Rewriting

Binary rewriting refers to the process of modifying one binary into another while one or more new instructions are optionally inserted to provide new features or behaviors. Binary rewriting methods can be categorized into two main classes: dynamic and static methods. Approaches [22, 40, 43] that belong to the first category transform stripped binaries that are loaded into memory while they are executing. However, they are not practical for fault observation on bare-metal devices due to the high performance overhead and special software/hardware requirements. HALucinator [26] is the state-of-the-art approach for dynamic binary instrumentation of bare-metal devices. In addition to the significant performance penalty, HALucinator only supports a small number of microcontrollers. HALucinator emulates firmware that use a Hardware Abstraction Layer

(HAL), and it is only applicable when the HAL is available to the analyst. Consequently, HALucinator is not mature enough for instrumentation in large-scale stripped firmware binaries. BinRec [17] is a dynamic binary rewriter that leverages multiple dynamic analysis techniques to lift binaries into LLVM IR. BinRec is built on top of S2E [25] and QEMU virtual machine [21] that do not support bare-metal firmware.

There are a number of static binary rewriting approaches that transform binaries before execution. These approaches differ from each other in how they transform binaries without breaking their functionality and semantics. Solutions such as Bistro [27] and STIR [56] redirect control flow from the original location to trampoline code containing new instructions. Trampoline-based rewriters are able to preserve application semantics after instrumentation, at the cost of considerable performance and memory penalties.

Uroboros [55] presents a set of heuristics for recognizing references among integer values and converting them into assembler labels in order to generate a relocatable assembly code. Rambler [54], built atop angr [49, 50], is a similar approach that improves Uroboros using a composition of static analyses and heuristics. Unfortunately, heuristics-based approaches suffer from false positives and negatives that result in broken reassembled binary. RetroWrite [28] and Egalito [59] provide an instrumentation method that uses relocation information which is only available in position independent codes. This is not a practical solution for firmware binaries that are statically linked. Multiverse [20] leverages a superset disassembling technique [38, 39, 58] and disassembles at each offset of the binary code to produce a superset of instructions. Multiverse binary rewriter is built on top of the disassembler to instrument all superset instructions. As noted by Miller et al. [41], superset disassembly has a substantial code size overhead (763% on SPECint 2006 benchmarks). Furthermore, experimental results [59] show that Multiverse does not support statically linked binaries.

All the above static approaches are designed and developed for the x86 architecture. RevARM [36] is the only static binary rewriter proposed for instrumenting ARM-based mo-

Table 4: A subjective comparison between μ SBS and state-of-the-art binary rewriting methods proposed for General Purpose (GP) computers, Mobile (M), and Bare-metal (B) devices. X and A denote x86 and ARM architectures respectively.

Objectives	Uroboros [55]	Ramblir [54]	RetroWrite [28]	Egalito [59]	Multiverse [20]	RevARM [36]	μ SBS
Target	GP	GP	GP	GP	GP	M	B
Architecture	X	X	X	X	X	A	A
w/o Heuristics	X	X	✓	✓	✓	X	✓
w/o Relocation	✓	✓	X	X	✓	✓	✓
w/o IR Lifting	✓	✓	✓	✓	✓	X	✓

ble applications. RevARM lifts binary code to a higher-level intermediate representation (IR) and performs the instrumentation procedure at that level. As pointed out by Dinesh et al. [28] lifting a binary to an IR usually misses application semantics and actual control flows since it is necessary to precisely model instruction set architecture (ISA) and extract type information from the binary file. Additionally, RevARM uses the Uroboros technique for differentiating references and integer values which is an impractical solution that is unable to work on non-trivial application binaries. Table 4 presents a subjective comparison of state-of-the-art binary rewriting approaches and μ SBS.

7 Discussion

In this section, we discuss the limitations in our system and shed some light for future work.

Supported microcontrollers. This paper focuses on a specific subclass of embedded microcontrollers running a single statically linked firmware—bare-metal firmware. Like all other static binary instrumentation methods, we do not handle any dynamically loaded code. Support for sanitizing such code requires dynamic instrumentation since such code can only be seen while the firmware is running.

Supported CPU architectures. The current implementation of μ SBS supports ARMv7-M architecture as the most widely used core for embedded systems [35]. Our platform-independent approach can support bare-metal firmware developed for other architectures like x86 with a small extra engineering effort since they are comparable or have more relaxed requirements for the binary instrumentation purpose [36]. For example, it is mainly required to change the assembly language of the rewritten and inserted instructions and mapping function in order to support x86 firmware.

Fuzzing. Although the current μ SBS implementation has

focused on the observation of faulty states due to the memory corruptions, it may be extended and integrated with fuzzing methods to uncover new bugs in bare-metal firmware. To be concrete, μ SBS sanitizer can be leveraged to improve the bug-finding ability of fuzzing methods [47, 63]. More specifically, we may guide the input generation process of the fuzzers towards triggering μ SBS sanitizer checks. Improvement on IoT fuzzing [31, 32, 62] is orthogonal to this paper, and we will leave it for future work.

Sanitization. μ SBS can potentially observe a wide array of memory corruptions by applying memory safety policies. The current implementation of μ SBS sanitization process is inspired by AddressSanitizer and Valgrind’s Memcheck policies. However, other sanitization techniques can be developed on top of μ SBS *binary instrumentation* component for observing faulty states caused by other types of memory corruption vulnerabilities. We leave such improvements to future work.

8 Conclusion

Memory corruption vulnerabilities are common in IoT firmware binaries and can lead to significant damage on bare-metal embedded devices that are increasingly intertwined with critical industrial and medical processes. In this paper, we have presented a concrete investigation of hardware security feature (i.e., MMU, MPU, and DEP) in a representative selection of IoT SoC families. Our analysis shows that the IoT fuzzing world lags behind the general-purpose world. We have also developed and demonstrated μ SBS, the first fully automatic approach for observing faulty states in bare-metal firmware. μ SBS uses a novel combination of static binary instrumentation and sanitization to validate memory accesses in a firmware binary, allowing for an improved fault observation mechanism. We evaluated μ SBS using a fault observation benchmark and 11 real firmware binaries. Our approach correctly sanitized all the firmware binaries with reasonable run-time over-head and size expansion while discovering the same set of vulnerabilities as the state-of-the-art. To motivate further research in this field and encourage reproducibility, we open-source μ SBS at <https://github.com/pwnforce/uSBS>.

Acknowledgments

This research is supported by the research fund of KU Leuven and imec, a research institute founded by the Flemish government. The work of the third author has been partially supported by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe. We are grateful to anonymous reviewers for assisting us with their helpful comments and criticisms.

References

- [1] ARMv7-M architecture reference manual. https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf. Accessed: February 2020.
- [2] BooFuzz source code repository. <https://github.com/jtpereyda/boofuzz>. Accessed: February 2020.
- [3] Capstone: The ultimate disassembler framework. <http://www.capstone-engine.org/>. Accessed: February 2020.
- [4] CVE-2017-0561. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0561>. Accessed: February 2020.
- [5] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. <https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016>. Accessed: February 2020.
- [6] Google Project Zero: Over The Air: Exploiting Broadcoms Wi-Fi Stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html. Accessed: February 2020.
- [7] Pwntools: CTF framework and exploit development library. <https://github.com/Gallopsled/pwntools/>. Accessed: February 2020.
- [8] Pyelftools: Parsing ELF and DWARF in Python. <https://github.com/eliben/pyelftools/>. Accessed: February 2020.
- [9] Quarkslab Lief project. <https://lief.quarkslab.com/>. Accessed: February 2020.
- [10] STM32Cube MCU Packages. <https://www.st.com/en/embedded-software/stm32cube-mcu-packages.html>. Accessed: February 2020.
- [11] STM32F401RE MCU. <https://www.st.com/en/evaluation-tools/nucleo-f401re.html>. Accessed: February 2020.
- [12] STM32F479I MCU. <https://www.st.com/en/microcontrollers-microprocessors/stm32f469-479.html>. Accessed: February 2020.
- [13] STM32F4DISCOVERY MCU. <https://www.st.com/en/evaluation-tools/stm32f4discovery.html>. Accessed: February 2020.
- [14] STM32L152RE MCU. <https://www.st.com/en/evaluation-tools/nucleo-l152re.html>. Accessed: February 2020.
- [15] WYCNINWYC application source code repository. https://github.com/avatartwo/ndssi8_wycinwyc. Accessed: February 2020.
- [16] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46. IEEE, 2019.
- [17] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: Dynamic binary lifting and recompilation. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2020.
- [18] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600. USENIX Association, 2016.
- [19] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860. USENIX Association, 2014.
- [20] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [21] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2005.
- [22] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 265–275. IEEE, 2003.
- [23] Derek Bruening and Qin Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 213–223, 2011.
- [24] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

- [25] Vitaly Chipounov, Volodymyr Kuznetsov, , and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [26] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2020.
- [27] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: binary component extraction and embedding for software security applications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2013.
- [28] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P’)*, 2020.
- [29] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.
- [30] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
- [31] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.
- [32] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150. USENIX Association, 2019.
- [33] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
- [34] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the winter 1992 USENIX conference*. USENIX Association, 1991.
- [35] Chung Hwan Kim, Taegy Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.
- [36] Taegy Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. RevARM: A platform-agnostic ARM binary rewriter for security applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 412–424. ACM, 2017.
- [37] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 423–427. Springer, 2008.
- [38] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2004.
- [39] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and George Portokalidis. GPU-Disasm: A gpu-based x86 disassembler. In *Proceedings of the international Information Security Conference*, pages 472–489, 2015.
- [40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, 2005.
- [41] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198. IEEE, 2019.
- [42] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

- [43] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Programming Language Design and Implementation (PLDI)*, 2007.
- [44] Majid Salehi, Danny Hughes, and Bruno Crispo. Microguard: Securing bare-metal microcontrollers against code-reuse attacks. In *Proceedings of the IEEE Conference on Dependable and Secure Computing (DSC)*, 2019.
- [45] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, 2002.
- [46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 309–318, 2012.
- [47] Kostya Serebryany. Sanitize, fuzz, and harden your c++ code. In *USENIX Enigma*. USENIX Association, 2016.
- [48] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2005.
- [49] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmallice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Sym. (NDSS)*, 2015.
- [50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 138–157. IEEE, 2016.
- [51] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1275–1295, 2019.
- [52] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2013.
- [53] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable use-after-free detection. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.
- [54] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
- [55] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642. USENIX Association, 2015.
- [56] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012.
- [57] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 299–308. ACM, 2012.
- [58] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecidable path. In *Proceedings of the Pacific-Asia Conference Knowledge Discovery and Data Mining (PAKDD)*, pages 273–285, 2014.
- [59] David Williams-King, Hidenori Kobayashi, Kent Williams-King, G. Elaine Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.
- [60] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2010.
- [61] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

- [62] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114. USENIX Association, 2019.
- [63] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

BlueShield: Detecting Spoofing Attacks in Bluetooth Low Energy Networks

Jianliang Wu^{1*}, Yuhong Nan^{1*}, Vireshwar Kumar¹, Mathias Payer², and Dongyan Xu¹

¹Purdue University, ²EPFL

{wu1220, nan1, viresh}@purdue.edu, mathias.payer@nebelwelt.net, dxu@cs.purdue.edu

Abstract

Many IoT devices are equipped with Bluetooth Low Energy (BLE) to support communication in an energy-efficient manner. Unfortunately, BLE is prone to spoofing attacks where an attacker can impersonate a benign BLE device and feed malicious data to its users. Defending against spoofing attacks is extremely difficult as security patches to mitigate them may not be adopted across vendors promptly; not to mention the millions of legacy BLE devices with limited I/O capabilities that do not support firmware updates.

As a first line of defense against spoofing attacks, we propose *BlueShield*, a legacy-friendly, non-intrusive monitoring system. BlueShield is motivated by the observation that all spoofing attacks result in anomalies in certain cyber-physical features of the advertising packets containing the BLE device's identity. BlueShield leverages these features to detect anomalous packets generated by an attacker. More importantly, the unique design of BlueShield makes it robust against an advanced attacker with the capability to mimic all features. BlueShield can be deployed on low-cost off-the-shelf platforms, and does not require any modification in the BLE device or its user. Our evaluation with nine common BLE devices deployed in a real-world office environment validates that BlueShield can effectively detect spoofing attacks at a very low false positive and false negative rate.

1 Introduction

An increasing number of IoT devices leverage Bluetooth Low Energy (BLE) to communicate in an energy-efficient manner. As one of the most popular protocols for IoT devices (e.g., smart locks, smart lights, or smart thermostats) [36], BLE will be empowering up to 5 billion devices by 2023 [4]. Given the popularity of BLE, there is an increasing concern about its security due to the discovery of threats that enable illegal device access, user fingerprinting, and inference of the device's sensitive information [13, 20, 26, 37]. While various approaches have been proposed to mitigate some threats [21, 33], most of them focus on the protection of the BLE device itself, less noticed here is the end-to-end communication between a *BLE device* and a *user device* in the network, where a spoofing attack is a major threat.

*The two authors contributed equally.

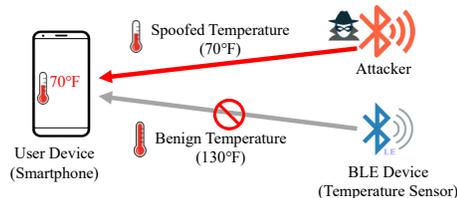


Figure 1: A typical BLE spoofing attack scenario.

Figure 1 presents a spoofing attack where an attacker device impersonates a benign BLE device (e.g., a temperature sensor) and feeds malicious data to the user device (e.g., a smartphone). Spoofing attacks are becoming critical, especially given that BLE devices are now widely integrated into security and privacy-sensitive scenarios [6, 18, 27], including manufacturing plants, medical and health care, and even physical-security monitoring.

The BLE specification (p. 2798 in [5]) provides a variety of authentication mechanisms that can potentially be employed to prevent spoofing attacks. Yet in practice, these mechanisms fail to serve their purpose due to two reasons: (1) A large proportion of BLE devices are limited by their I/O capabilities, and hence cannot employ any secure authentication mechanism. Unsurprisingly, a recent study has revealed that nearly 80% of existing BLE devices communicate with their user devices in plaintext without any authentication [30]. (2) Even in BLE devices employing different security mechanisms, there exist a variety of attack surfaces, at both BLE protocol-level [49] and application-level [37], which can be exploited by malicious attackers to launch spoofing attacks.

Unfortunately, deploying a software-based solution (i.e., firmware updating of BLE devices and software patching on user devices) to prevent spoofing attacks is challenging due to three practical challenges: (1) Software patching cannot defend against zero-day spoof-enabling vulnerabilities [49]. (2) Firmware updates to mitigate the vulnerabilities of BLE devices cannot be uniformly created and applied at a large scale by the wide range of BLE device vendors [21]. (3) Even worse, there are millions of (already deployed) legacy BLE devices in the field that do not support firmware updates due to their limited I/O capabilities.

An Out-of-the-Box Defense. To address the aforementioned challenges, we propose *BlueShield*, a vulnerability-agnostic, legacy-friendly and non-intrusive monitoring system

to protect BLE devices against spoofing attacks. The design of BlueShield is motivated by the critical observation that, in a spoofing attack, the attacker must broadcast *advertising packets* (containing the BLE device’s identity) on BLE *advertising channels* to make itself discoverable by the user device. As such, BlueShield detects the spoofing attack by distinguishing the advertising packets transmitted by the attacker and the benign BLE device.

To collect all over-the-air packets on BLE advertising channels, BlueShield sets up a monitoring infrastructure covering the physical environment where BLE devices are deployed. Then, BlueShield inspects a combination of *cyber* and *physical* features extracted from the advertising packets of the target BLE device, and raises an alarm (indicating the presence of a spoofing attack) if it detects any anomaly. Here, while the cyber features include the advertising pattern and state transitions of the BLE device, the physical features include advertising interval, the RF (radio frequency) signal frequency offset, and the RF signal strength (Section 4.1).

The effectiveness of BlueShield against spoofing attacks stems from its two novel mechanisms: (1) To detect an advanced attacker, such as a software-defined radio (SDR)-enabled attacker with the capability to mimic all the physical features, BlueShield employs a *randomized channel switching* mechanism (Section 4.3.1) to collect physical features of the monitored device across different advertising channels. This mechanism brings forth a “moving target defense” so that the attacker cannot reliably predict and mimic the correct values of the monitored features to evade the detection. (2) BlueShield also employs a *selective inspection* mechanism (Section 4.3.2) for triggering the alarm: For each received advertising packet, BlueShield inspects one or more of the “physical” features which are selected based on the determined “cyber” features of the BLE device. Then, an anomalous value is detected by comparing the current physical feature values with their valid ranges. This mechanism effectively mitigates the occurrence of false alarms due to unintentional interference in physical features, while performing in-time detection of spoofing attacks with good precision. As such, the two aforementioned mechanisms ensure that the selected features (even individually) cannot be easily imitated or controlled by an attacker, and hence can be utilized together to robustly detect spoofing attacks.

We evaluate BlueShield on nine BLE devices (including temperature sensors and smart locks) covering different types of devices widely deployed in security/safety-sensitive environments, such as smart homes, museums, and manufacturing plants. We launch various spoofing attacks with different strategies and from different locations in a real-world, noisy office environment. Our evaluation results demonstrate that BlueShield can effectively detect spoofing attacks with an average success rate of 99.28% on our tested BLE devices. Moreover, even in a usage-heavy scenario BlueShield only introduces less than one false alarm a week.

Compared with defenses in the existing literature [21, 28, 37], BlueShield provides four practical advantages: (1) Since the monitoring infrastructure captures and analyzes all traffic on advertising channels, BlueShield supports concurrent monitoring of many BLE devices. (2) BlueShield is deployable in all BLE networks regardless of the capability of the BLE device and the version of its BLE implementation. (3) BlueShield is fully transparent to its deployed environment, i.e., it does not introduce any interference or energy overhead to the BLE devices and user devices. (4) BlueShield does not require any firmware modification or reverse engineering of the BLE devices or user devices.

Our major contributions are summarized as follows:

- We propose BlueShield, a generic, device-agnostic monitoring framework to detect spoofing attacks in BLE networks using a novel combination of selected cyber and physical features of the BLE devices.
- We devise a set of mechanisms to utilize the selected features for detecting spoofing attacks. These mechanisms not only ensure the robustness of BlueShield against an advanced SDR-enabled attacker, but also provide good effectiveness with very low false alarms.
- We have implemented BlueShield using off-the-shelf components and thoroughly evaluated its effectiveness and efficiency in a real-world environment.

2 Background and Motivation

2.1 BLE Basics

BLE is typically deployed in a network that requires an energy-constrained low-cost device (e.g., temperature sensor) to record an attribute/data value (temperature) and communicate the data to a user device (e.g., smartphone) over the wireless medium. In BLE, three radio frequency (RF) channels (channel-37, channel-38, and channel-39) are utilized for advertising and initializing the connection with the user device and are called *advertising channels*. The other channels are utilized for communicating data and are called *data channels*. The typical communication procedure between the BLE device and the user device can be broadly classified into four steps: advertising, connecting, pairing, and accessing.

Advertising. The BLE device publicizes its presence and facilitates its discovery by periodically broadcasting *advertising packets* on the three advertising channels. Each advertising packet contains a unique identifier of the BLE device and the information about services provided by it.

Connecting. The user device scans the three advertising channels and discovers the BLE device using its advertising packet. If the user device intends to establish a connection, it sends a *connection request* packet. If the BLE device accepts the connection, the BLE device and the user device start communication at data channels. We note that during

the connection state, most of the BLE devices stop public advertising and communicate only with the connected user devices. However, some BLE devices may keep advertising even during the connection state.

Pairing. The BLE specification provides multiple pairing methods to establish a secure channel for communication. Through the pairing procedure, the BLE device and the user device can generate a shared key that can be utilized for encryption and authentication of the communicated payloads.

Accessing. After establishing the connection in each session, the user device utilizes services provided by the BLE device by accessing corresponding data values, e.g., a temperature value provided by a smart temperature sensor.

2.2 Spoofing Attacks in BLE Networks

Reasons for Successful Spoofing Attacks. Recent studies have revealed that BLE networks expose various attack surfaces that can be exploited by an adversary to launch spoofing attacks [2, 33, 37, 46, 47]. Since around 80% of BLE devices do not perform secure pairing and communicate data without employing a secure authentication mechanism [30], they are vulnerable to spoofing attacks. Even for the remaining devices that employ authentication mechanisms after pairing securely, there exist zero-day vulnerabilities across different layers of the BLE stack due to flaws in the design and implementation of the BLE specification [1, 2, 46, 47]. All such vulnerabilities can be easily exploited by the attackers to perform spoofing attacks against authentication-enabled devices. In fact, we have also discovered a protocol-level spoof-enabling vulnerability that affects major platforms running the most recent version of iOS and Android [46]. We have responsibly reported it to Apple and Google¹. Apple has acknowledged the reported vulnerability and assigned CVE-2020-9970 [14] to it.

Limitations of Existing Solutions. A straightforward solution to prevent spoofing attacks could be to upgrade the BLE devices with advanced security features, and in the meantime, adopt security patches to fix vulnerabilities. However, such solutions will not work well in practice due to the extremely fragmented nature of the BLE ecosystem. Specifically, the case-by-case firmware updates and security patches may not be preferably adopted by different vendors at a large-scale, since vendors always tend to sacrifice security for utility. Moreover, it is extremely difficult for those low-end, legacy BLE devices (e.g., beacons) with limited I/O capabilities to adopt any firmware updates or security patches. Furthermore, a solution that fails to ensure backward compatibility with the legacy BLE devices is unlikely to be deployed widely. Hence, these limitations of existing solutions call for a practical and legacy-friendly defense to effectively alleviate the threat of spoofing attacks in a more fundamental way.

¹While Google has confirmed the vulnerability, we have been told that another researcher reported a similar vulnerability three days earlier than us.

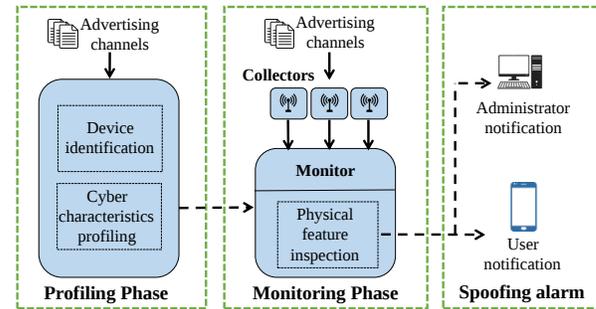


Figure 2: Architecture of BlueShield.

3 Overview of BlueShield

BlueShield is a monitoring framework for detecting spoofing attacks and alarming the user in a *stationary* BLE network. Specifically, it detects the footprint of spoofing attacks by noticing anomalies in the unique combination of cyber and physical features (Section 4.1) of advertising packets containing the BLE device’s identity. Since the nature of a successful spoofing attack requires the attacker to broadcast the spoofed advertising packets *before* the appearance of a victim user, such activities expose the attacker in advance and provide a great deal of opportunity for BlueShield to detect before any actual damage is inflicted on the victim user.

Architecture and Workflow. Figure 2 presents the architecture of BlueShield. For each BLE device in an indoor environment that BlueShield aims to protect, the workflow goes through two phases, namely the *profiling phase* and the *monitoring phase*. The profiling phase (Section 4.2) involves the offline procedures to determine the BLE device’s authentic protocol-level cyber features, such as MAC address, advertising data, and advertising pattern.

The monitoring phase (Section 4.3) includes runtime procedures to detect spoofing attacks using an infrastructure consisting of three *collectors* and one *monitor*. The three collectors follow a randomized channel switching mechanism (Section 4.3.1) through which each collector is randomly assigned to collect the advertising packets on one of the three advertising channels. Since the authentic value of the physical features (e.g., RF signal’s strength) of the advertising packets would be different at different channels, this mechanism ensures that even an attacker with the capability to mimic all the physical features, cannot trick all the three collectors with correctly imitated values at the same time.

After obtaining the feature values of the advertising packets gathered by the three collectors, the monitor employs a selective inspection mechanism (Section 4.3.2) which exploits the cyber features obtained in the profiling phase to select appropriate physical features for an effective runtime inspection. Thereafter, the monitor determines the valid range of the selected physical features over each advertising channel. Finally, if the monitor discovers that the obtained feature val-

ues lie outside their valid ranges, it alarms the user and/or the network administrator about a detected spoofing attack.

Adversary Model. BlueShield is subject to the following assumptions: We assume that the attacker aims to impersonate a target BLE device and launch a spoofing attack against the user device. The attacker is capable of passively sniffing the traffic, and actively crafting and transmitting spoofed packets on BLE channels (e.g., using a customized BLE device). An advanced attacker may even employ advanced evasion techniques (e.g., with an SDR) and try to mimic the features of the wireless signals of the target BLE device. Besides, the attacker can also exploit the vulnerabilities of the authentication mechanism between the BLE device and user device [2]. However, we assume that the attacker cannot fully control the target BLE device either physically (e.g., remove/replace the target device with his own) or remotely (e.g., by corrupting its firmware). Also, we do not consider the attacker that uses jamming, as the detection of jamming is sufficiently covered by existing research [32, 48] which are orthogonal to BlueShield. Lastly, for BlueShield, we assume that the monitoring infrastructure for collecting the advertising packets is secure, i.e., it cannot be compromised by the attacker.

Scope of Detection. BlueShield focuses on detecting spoofing attacks that target *stationary BLE devices in indoor environments*. As such, BlueShield covers a wide range of BLE usage scenarios (e.g., smart homes, museums, and manufacturing plants) [17]. Specifically, a recent report [22] showed that 15 out of the 18 (83.3%) most popular IoT devices in smart homes are stationary. We note that spoofing detection in outdoor environments or for movable devices (e.g., fitness trackers) is out of the scope of BlueShield.

4 Detailed Design

In this section, we first describe the physical and cyber features inspected by BlueShield. We then elaborate on how these features are used in the profiling and monitoring phases for enabling an effective detection of spoofing attacks.

4.1 Inspected Features

4.1.1 Physical Features

Advertising Interval (INT). We define INT as the time between two consecutive advertising packets on the *same* advertising channel. In spoofing attacks, if both the BLE device and attacker are broadcasting advertising packets, the real-time INT can be lower than the expected INT value, and hence can be utilized to detect the spoofing attacks.

Carrier Frequency Offset (CFO). The CFO of a given device depends on its RF circuit’s imperfections which induce a unique mismatch/offset between the designated and the actual carrier frequency of the transmitted signal. As a result,

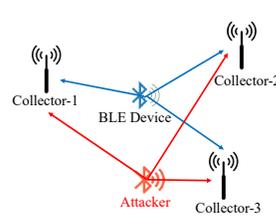


Figure 3: Locations of the BLE device, the collector, and the attacker.

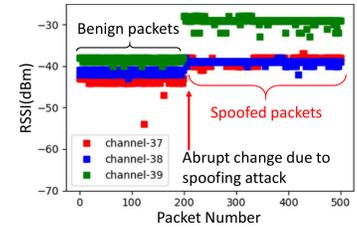


Figure 4: Observed RSSI values under spoofing attacks (with typical channel interference).

CFO can be utilized as a fingerprint of a wireless device [7, 39]. In the case of spoofing attacks, the CFO extracted from advertising packets can be used to differentiate the packets generated by the benign BLE device and the attacker.

Received Signal Strength Indicator (RSSI). RSSI is a numerical estimate of the signal power in the received RF signal, which depends on the distance and the channel interference between a transmitter and receiver [19]. In the prior art, RSSI has been widely used for fingerprinting the location of a wireless device [19, 24, 50]. In the case of spoofing attacks, since the locations of the BLE device and the attacker are different as illustrated in Figure 3, the RSSI values of the benign and spoofed advertising packets can be utilized to discover spoofing attacks. For instance, as shown in Figure 4, the abrupt change in RSSI values of advertising packets indicates the presence of a spoofing attack.

4.1.2 Cyber Features

Although the physical features enable BlueShield to detect spoofing attacks, they share some inherent limitations. Specifically, the INT feature will only work for BLE devices which persistently transmit advertising packets before and after their connections with user devices. In the meantime, as shown in Figure 4, like any feature of wireless signals, the values of the CFO and RSSI can change due to unintentional signal interference (e.g., human movements). Hence, continuously inspecting these two features will trigger a significant number of false alarms. To overcome these challenges, BlueShield employs additional cyber features that are determined by device-specific BLE implementation, to support the usage of the aforementioned physical features.

Advertising Pattern. According to the BLE specification, a BLE device can illustrate one of the two advertising patterns: intermittent and persistent. A BLE device with an *intermittent advertising pattern* stops advertising after connecting to a user device. A BLE device with a *persistent advertising pattern* continues to advertise even after connecting to a user device.

Operation State. The BLE specification defines that every BLE device stays active in one of the two states: advertising and connection. In the *advertising state*, the BLE device

Table 1: An illustrative sample of characteristics of a BLE device recorded during the profiling phase.

Characteristic	Value
Device ID & Name	1, n097w
MAC Address	0xD1 76 A3 1A F4 7F
Advertising Data	0x06 09 4E 30 39 37 57
Advertising Pattern	Intermittent
Lower Bound of INT	1280 ms

makes itself discoverable by broadcasting advertising packets periodically. When the BLE device receives a connection request packet from the user device, the BLE device connects to the user device and makes the transition to the *connection state*. From the connection state, the BLE device returns to the advertising state if it does not communicate with the user device for a specified timeout period or disconnects from the user device. We highlight that for a BLE device with an intermittent advertising pattern, the advertising-to-connection state transition can be detected by observing a connection request packet on an advertising channel.

The two cyber features mentioned above naturally support BlueShield to selectively choose one or more of the physical features for detecting spoofing attacks. Specifically, for BLE devices with the persistent advertising pattern, inspecting only the INT values is sufficient to detect potential spoofing attacks. For other BLE devices that follow the intermittent advertising pattern, the attacker can stop the benign BLE device from advertising by connecting to it. Then the attacker can start transmitting spoofed advertising packets with the same advertising period as the benign BLE device. To detect such an attacker, the CFO and RSSI features can be used once BlueShield detects that there is an advertising-to-connection state transition in the BLE device.

4.2 Profiling Phase

Now we describe the procedures performed in the offline profiling phase of BlueShield. To obtain the data-of-interest of a target BLE device, the monitor records and analyzes the advertising packets of the BLE device. First, from the packet content, the monitor extracts the device name, the MAC address, and the advertising data which can be utilized to identify packets transmitted by the BLE device. We note that although some BLE devices employ address randomization to anonymize their identity (p. 2198 in [5]), some of the fields (e.g., device name) in their packets remain unchanged and can be used to relate the packets to the BLE device [13, 20].

The monitor then computes the INT value by subtracting the time-of-arrival of the current advertising packet from that of the previous advertising packet. As defined by the BLE specification (p. 2750 in [5]), the observed INT is equal to a fixed *advertising period* plus a random delay between 0 and 10 ms. To this end, the monitor calculates the lower bound

(a) Authentic RSSI values (in dBm) corresponding to the BLE device at different advertising channels.

Channel	Collector 1	Collector 2	Collector 3
Channel 37	-60.3	-48.6	-39.2
Channel 38	-63.5	-45.2	-35.8
Channel 39	-58.1	-46.3	-36.9

(b) Assigned channel and corresponding RSSI value during different time periods at each collector.

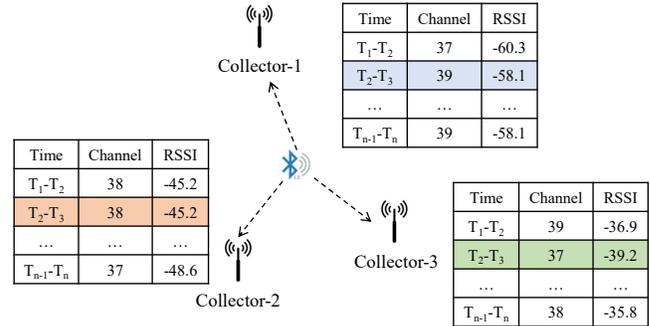


Figure 5: An illustration of the randomized channel switching.

of INT as the shortest observed INT minus 10 ms, which represents the lowest possible interval between any two advertising packets on the same channel. Further, the monitor determines the authentic advertising pattern (i.e., persistent or intermittent) by checking if it observes the BLE device’s advertising packets after connecting to the BLE device. Finally, the monitor stores the determined characteristics of the BLE device along with an assigned device identifier (ID) as shown in Table 1. The deployment details regarding the profiling phase are discussed further in Section 7.

4.3 Monitoring Phase

After the profiling phase, BlueShield is ready to detect spoofing attacks in its runtime monitoring phase.

4.3.1 Feature Collection

BlueShield faces the critical challenge of detecting an attacker which can attempt to: (1) start the spoofing attack at any time, (2) transmit spoofed advertising packets at any of the three advertising channels, and (3) hide by trying to mimic the exact values of the selected physical features. To tackle these challenges, BlueShield first places the three collectors at three different locations to comprehensively cover the monitored environment in space, and then ceaselessly records all advertising packets transmitted at all three advertising channels using the following mechanism.

Randomized Channel Switching. In addition to the spatial diversity of the collectors, BlueShield introduces randomness in the monitoring schedule of each collector. To achieve this, BlueShield assigns the three advertising channels to the three

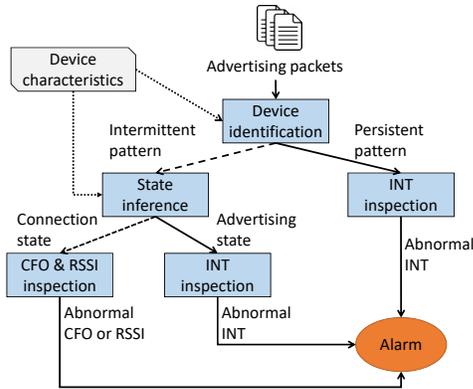


Figure 6: Usage of the cyber-physical features for selective inspection of the physical features in BlueShield.

collectors with a uniformly random distribution. After a very short random interval of time (which is significantly shorter than the BLE device advertising interval), the monitor shuffles and re-assigns channels to the collectors. Since the values of each physical feature (i.e., RSSI and CFO) in different advertising channels are different, the attacker will not be able to precisely predict the expected feature values at runtime. This randomized collection makes BlueShield robust against advanced attackers (e.g., an SDR-enabled attacker) that try to mimic all the monitored physical features at the same time.

Figure 5 demonstrates a running example of this mechanism by presenting the randomly assigned channel and the expected RSSI value at each collector during different periods. As such, since the channel assignment to the collectors is random, an attacker will not be able to accurately guess the current “time-channel-collector” mapping. In other words, the attacker will not know the exact RSSI value that it needs to mimic for a particular collector at a specific time. We provide an analytical evaluation of this mechanism in Section 6.2.2.

4.3.2 Runtime Selective Inspection

After retrieving the advertising packets and their features from the collectors, the monitor proceeds with a runtime selective inspection of these features for each BLE device. As shown in Figure 6, the cyber features of the target BLE device allow BlueShield to adaptively employ the appropriate inspection mechanism over the physical features. This selective inspection significantly lowers the false alarms caused by signal interference as will be further discussed in Section 6.2.1.

Now we describe the inspection mechanisms as follows.

INT Inspection. The monitor proceeds with this inspection mechanism for each received advertising packet from the BLE device. Recall that by definition, the INT between any two advertising packets must always be more than the *lower bound* of INT. Hence, if the runtime computed INT value is less than the lower bound of INT, the monitor considers it an anomaly and raises an alarm.

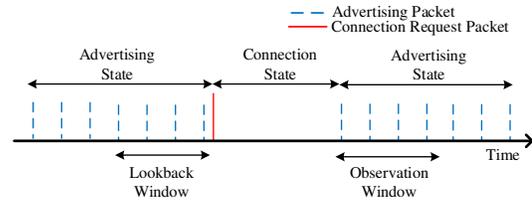


Figure 7: Illustration of the lookback and observation windows in CFO and RSSI inspection.

CFO and RSSI Inspection. BlueShield ceaselessly records the CFO and RSSI values of the advertising packets. When the CFO and RSSI inspection is triggered, BlueShield utilizes these values and proceeds through the following steps. As shown in Figure 7, for a BLE device with an intermittent advertising pattern, we define the *lookback window* as the duration of time T_l (with N_l packets) before the advertising-to-connection state transition, and the *observation window* as the duration of time T_o (with N_o packets) after the connection-to-advertising state transition. In BlueShield, after observing a connection request packet, the monitor triggers the CFO and RSSI inspection for advertising packets received from each collector in each of the three advertising channels. The monitor first utilizes CFO and RSSI values of advertising packets in the lookback window to compute their valid ranges, then it inspects these values of advertising packets in the observation window. If the monitor detects an anomaly in either of these two features, it raises an alarm. Below we elaborate on the process of CFO inspection. The RSSI inspection follows similar steps as shown in Appendix A.

The CFO values corresponding to a BLE device follow a Gaussian distribution [43]. Hence, when the mean and standard deviation of CFO values are denoted by μ_0 and σ_0 , respectively, the probability distribution function of CFO values can be computed as $f_c(x_i) = \frac{1}{\sigma_0\sqrt{2\pi}} \cdot e^{-(x_i-\mu_0)^2/2\sigma_0^2}$, where x_i denotes a CFO sample. In BlueShield, using the N_l CFO values of advertising packets in the lookback window, the monitor computes μ_0 and σ_0 , and then sets their values in the above function. Now if the advertising packets in the lookback and observation windows are generated by the same BLE device, the CFO values of advertising packets in the observation window must statistically follow the above distribution. To verify this, the monitor computes the negative log-likelihood of the CFO values in the observation window, i.e., $L_c = \frac{1}{N_o} \sum_{i=1}^{N_o} -\log f_c(x_i)$. If the log-likelihood, L_c is below a CFO inspection threshold denoted by τ_c , i.e., $L_c \leq \tau_c$, the CFO values are considered to belong to the benign BLE device. Here, τ_c is a design parameter in BlueShield which determines the valid range of CFO values in the observation window. However, if the log-likelihood exceeds the CFO inspection threshold, i.e., $L_c > \tau_c$, the monitor considers it an anomaly and triggers an alarm indicating a spoofing attack. The impact of this parameter is further discussed in Section 6.

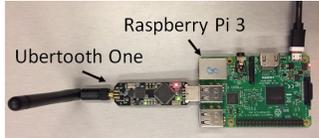


Figure 8: Prototype of a collector utilized in BlueShield.

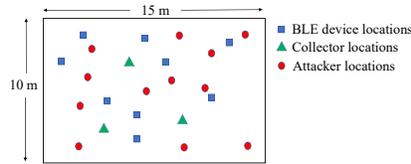


Figure 9: Locations of collectors, BLE devices and attackers within the office.



Figure 10: BLE devices used in our experiments.

5 Implementation

BlueShield can be readily implemented using low-cost, off-the-shelf platforms. We implemented the collector using an Ubertooth One radio [41] connected to a Raspberry Pi running Linux 4.14 (Figure 8). The total cost for such a collector is around \$100. We note that when a collector is deployed on a custom-designed platform, the per-unit cost could be less than \$5 for a BLE module [40]. The Ubertooth first captures the packets on advertising channels. Then, to retrieve the physical features, we modified the Ubertooth firmware to provide CFO and RSSI values for each received packet. Such customization is feasible as Ubertooth is an open platform for Bluetooth research and development. Finally, the Raspberry Pi communicates the packets along with their relevant features to the monitor. We implemented the monitor on an Ubuntu 18.04 Desktop PC. At the monitor, all processes including interacting with collectors, parsing the received information from collectors, and runtime inspection mechanisms were implemented with ≈ 3 K lines of Python code.

User Notification. The design of BlueShield supports notification of detected spoofing attacks. To demonstrate this use case, we have developed an Android application that employs a push-based mechanism to receive notifications from BlueShield’s monitor through a secure HTTPS connection (Appendix B). The user can register to BlueShield’s notification service by installing and configuring the app. Then, whenever BlueShield detects a spoofing attack, the user receives the notification with relevant information, such as the targeted BLE device’s name and MAC address.

6 Evaluation

6.1 Experiment Setup

Deployment Environment. We evaluated the detection performance of BlueShield in a real-world environment: a $15\text{m} \times 10\text{m}$ office hosting multiple graduate students in 20 cubicles. We divided the office space into grids of $1\text{m} \times 1\text{m}$. We deployed BlueShield by placing the three collectors at selected grid locations within the office as shown in Figure 9. The office presents a typically noisy and challenging indoor environment for determining the detection performance of BlueShield. By recording RF signals within the reception

range of the collectors, we discovered significant channel interference from 30 other Bluetooth/BLE-equipped devices (sensors, headsets, smartphones, and laptops), dozens of Wi-Fi access points and a microwave². We also observed that abrupt movements of students within the office significantly altered channel conditions in the monitored environment.

Device Selection. To exhaustively evaluate BlueShield, we utilized nine different BLE devices which are shown in Figure 10. These BLE devices cover the mainstream BLE applications (e.g., temperature sensor, lock, and smoke detector) and popular manufacturers (e.g., Nest, August, and Eve) with a variety of Bluetooth chips (e.g., DA14580 and nRF51822). As shown in Figure 9, we randomly chose nine different locations within the office to place these BLE devices.

Attack Simulation. To carry out different types of attacks, we utilized four attacker platforms, a Dell Latitude 5480 laptop [15], a CSR 4.0 BT dongle [11], an HM-10 developmental board [25], and a CYW920735 developmental board [12]. These platforms were selected because they provide ease of access and programmability, and they utilized different transmit power values. Besides, to thoroughly evaluate the performance of BlueShield, we launched a variety of spoofing attacks from 12 different locations, some at the center and some at edges of the office (Figure 9). Further, to enrich the evaluation of the effectiveness of the CFO inspection, we utilized different copies of the same BLE device as attackers (in addition to the four attacker platforms). For the unbiased evaluation of the RSSI inspection, we utilized the same BLE device as the benign BLE device and the attacker, and collected its advertising packets by placing it at different locations within the office environment.

Experimental Data. For each BLE device, benign advertising packets were collected for 48 hours (throughout day and night). For each attacker platform placed at each location, spoofed advertising packets were collected for around 15 minutes. In total, we collected 5,507,978 advertising packets which are comprised of 80.7% benign advertising packets and 19.3% spoofed advertising packets. This data was utilized as the ground truth for our evaluation.

²We only saved and analyzed the data of BLE devices deployed by us. We did not record any data from other devices within the office environment.

Table 2: BlueShield’s detection performance against spoofing attacks (FP and FN are presented in %).

Device ID	Device Name	Advertising Period (s)	Observation Window (s)	INT		CFO		RSSI		Overall	
				FP	FN	FP	FN	FP	FN	FP	FN
1	Nest Protect Smoke Detector	1.28	3.84	0.00	0.00	0.80	0.00	0.97	5.84	1.76	0.00
2	Nest Cam Indoor Camera	0.15	0.45	0.00	0.00	1.38	17.74	3.59	21.15	4.92	3.69
3	SensorPush Temperature Sensor	1.28	3.84	0.00	0.00	0.56	4.46	1.43	5.22	1.98	0.23
4	Tahmo Temp Temperature Sensor	2.00	6.00	0.00	0.00	0.64	0.00	1.32	22.94	1.95	0.00
5	August Smart Lock	0.30	0.90	0.00	0.00	1.12	4.85	1.26	1.60	2.37	0.08
6	Eve Door&Window Sensor	1.28	3.84	0.00	0.00	0.77	8.17	1.64	1.46	2.40	0.12
7	Eve Button Remote Control	1.28	3.84	0.00	0.00	0.98	1.41	1.18	3.00	2.15	0.04
8	Eve Energy Socket	0.15	0.45	0.00	0.00	0.60	1.67	0.85	1.55	1.44	0.03
9	Ilumi Smart Light Bulb	0.10	0.30	0.00	0.00	0.88	14.28	1.48	15.73	2.35	2.25
Average		0.87	2.61	0.00	0.00	0.86	5.84	1.52	8.72	2.37	0.72

6.2 Overall Effectiveness

The detection performance of BlueShield is evaluated in terms of two metrics: false positive (FP) and false negative (FN) misclassifications. A false positive refers to the error in which BlueShield generates a false alarm for a spoofing attack after inspecting benign advertising packets generated by the benign BLE device. A false negative refers to the error in which BlueShield fails to detect a spoofing attack after analyzing spoofed advertising packets from the attacker.

We highlight that the INT inspection, and the CFO and RSSI inspection are exclusively employed to detect spoofing attacks under different scenarios (Figure 6). The INT inspection readily uncovers an attack in which the attacker transmits spoofed advertising packets while the BLE device is broadcasting benign advertising packets. In such a scenario, the INT inspection does not introduce any FP and causes negligible FN in detecting attacks (Appendix C). However, the INT inspection cannot be used to detect an advanced attack where the attacker first suppresses the broadcast of benign advertising packets by connecting to the BLE device whose advertising pattern is intermittent, and then transmits spoofed advertising packets with the same advertising period. BlueShield employs the CFO and RSSI inspection to detect this type of attack. Since BlueShield raises an alarm for a spoofing attack when either CFO inspection or RSSI inspection detects an anomaly, BlueShield significantly reduces FN at the cost of a slight increase in FP. Theoretically, since these inspection mechanisms employ independent features, the overall FN of BlueShield can be calculated as $FN_{BlueShield} = FN_{CFO} \times FN_{RSSI}$. Also, the overall FP generated by BlueShield can be calculated as $FP_{BlueShield} = FP_{CFO} + FP_{RSSI} - FP_{CFO} \times FP_{RSSI}$.

6.2.1 Summary

As shown in Table 2, BlueShield achieves an average of 2.37% FP rate and 0.72% FN rate on our tested BLE devices. The results are obtained when the inspection thresholds τ_c and τ_r in the CFO and RSSI inspection are set to 3 and 5 respectively. The numbers of advertising packets in the lookback window (N_l) and observation window (N_o) are set to 100 and 3, re-

spectively. This means BlueShield is set to report a potential spoofing attack within three advertising periods, resulting in an average detection time of 2.61s based on the corresponding advertising period (see columns 3 and 4 in Table 2). Note that, we present our results with these reasonable values of parameters to illustrate a setting where BlueShield provides low FP values while detecting spoofing attacks against all nine BLE devices. BlueShield can also be employed with device-specific parameters with minimal effort to achieve an appropriate trade-off between FP and FN in a real-world deployment. For example, as shown in the receiver operating characteristic (ROC) curve in Figure 12, when the true positive decreases (i.e., FN increases), FP also decreases.

False Positive. As shown in Table 2, while the INT inspection does not introduce any FP, the average FP triggered by the CFO and RSSI inspection are 0.86% and 1.52% respectively. Overall, BlueShield achieves an average FP of 2.37%. To better illustrate the real-world impact of the FP rate, we consider a user device that makes five connections per day with the light bulb with Device ID = 9. We note that this is already a heavy usage scenario based on an IoT device usage frequency report [44]. As shown in Table 2, BlueShield achieves an average FP of 2.35% for this BLE device within an observation window of 0.3s. Hence, BlueShield will only cause an average of $5 \times 2.35/100 = 0.1175$ false alarm per day, which implies that there will be less than one false alarm during a one-week deployment. However, we point out that if we ceaselessly monitor the physical features of all packets transmitted by the BLE device, we would encounter $\frac{60 \times 60 \times 24 \times 2.35/100}{0.3} = 6768$ false alarms per day. Fortunately, BlueShield employs the selective inspection mechanism so that the CFO and RSSI inspection are triggered only when the BLE device makes an advertising-to-connection state transition. This mechanism drastically reduces such false alarms. Figure 11 further illustrates the impact of the selective inspection mechanism on the number of false alarms at different FP rates.

False Negative. In Table 2, we observe that while the INT inspection detects spoofing attacks quite robustly, the CFO and RSSI inspection achieve an average FN of 5.84% and 8.72% in detecting spoofing attacks. Overall, BlueShield achieves

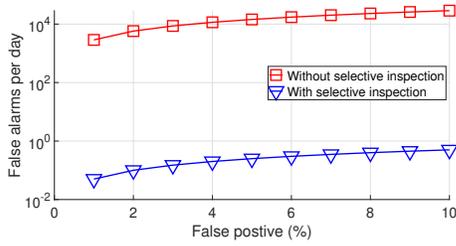


Figure 11: Number of false alarms per day with and without selective inspection mechanism in BlueShield.

an average FN of 0.72% which implies a detection rate of 99.28%. Further, we point out that most BLE devices force a disconnection after a certain time-out (e.g., 30 s). Therefore, to incessantly suppress benign advertising packets and replace them with spoofed advertising packets, an attacker may need to trigger multiple connections with the BLE device. As a result, the attacker is exposed to multiple CFO and RSSI inspections by BlueShield. For instance, if the attacker triggers three connections during a spoofing attack, BlueShield can fail to detect the attack with an average FN of only $3.7 \times 10^{-7}\%$ ($= 0.0072^3$).

6.2.2 Robustness against Advanced Attacker

As mentioned earlier, BlueShield’s randomized channel switching (Section 4.3.1) ensures that even for an advanced attacker which can precisely mimic the values of all physical features, there is barely any chance to evade detection. We evaluate the detection performance of BlueShield against such attacks as follows: In the randomized channel switching mechanism, the number of ways in which the monitor can assign N_c collectors the task to record and cover all the three advertising channels is given by $N_c! \binom{N_c-1}{3-1} = \frac{N_c!(N_c-1)(N_c-2)}{2}$. Hence, the probability with which an attacker can correctly guess the channel assignments over N_o advertising intervals within the observation window is given by $(\frac{2}{N_c!(N_c-1)(N_c-2)})^{N_o}$. For the three collectors (i.e., $N_c = 3$) recording the feature values over three advertising intervals (i.e., $N_o = 3$ as shown in Table 2), the probability of correct guesses by the attacker can be calculated as $(\frac{1}{6})^3 \approx 0.46\%$. As a result, the average detection rate of a spoofing attack launched by this type of attack over *one observation window* can be calculated as 99.54%. To this end, we conclude that the randomized channel switching enables BlueShield to effectively detect attackers that try to mimic all physical features used for detection.

6.3 Effectiveness of Individual Features

INT Feature. In the INT inspection, an anomaly is detected when the observed INT is less than the lower bound of INT. Since each of the benign BLE device’s INT is always larger

Table 3: Effectiveness of the CFO inspection in differentiating packets transmitted by different platforms.

Device ID	FP	FN				Average
		Dev-1 ¹	Dev-2 ²	Dongle ³	Laptop ⁴	
1	0.80	0.00	0.00	0.00	0.00	0.00
2	1.38	0.00	5.41	64.46	0.00	17.74
3	0.56	0.00	0.00	17.84	0.00	4.46
4	0.64	0.00	0.00	0.00	0.00	0.00
5	1.12	0.00	18.97	0.41	0.00	4.85
6	0.77	0.00	15.17	17.50	0.00	8.17
7	0.98	0.00	0.00	5.64	0.00	1.41
8	0.60	0.00	0.02	6.67	0.00	1.67
9	0.88	0.00	50.80	0.00	6.33	14.28
Average	0.86	0.00	10.04	12.50	0.70	5.84

1. HM-10 Bluetooth chip with expansion shield.
2. CYW920735Q60EVB-01 evaluation kit.
3. CSR 4.0 Bluetooth USB adapter.
4. Dell Latitude 5480 with built-in Qualcomm Bluetooth chip.

Table 4: CFO inspection results while using different copies of Nest Protect smoke detector as the BLE device and attacker.

	FP	FN		
		Copy-1	Copy-2	Copy-3
Copy-1	0.00	N/A	16.41	0.00
Copy-2	3.85	12.32	N/A	0.06
Copy-3	0.96	0.00	0.00	N/A

than the calculated lower bound of INT, by definition, there cannot be any FP in the INT inspection. Regarding the FN, recall that every broadcast of a spoofed advertising packet by the attacker results in an observed INT value that is smaller than the lower bound of INT. Hence, by noticing such an anomaly, the INT inspection detects the spoofing attack with negligible FN. Due to space limitation, we leave the detailed theoretical analysis about the FN in Appendix C.

CFO Feature. Through our extensive experiments, we validate two critical characteristics of the CFO feature. (1) The CFO value of a BLE device is not affected by changing its relative location to collectors. This is because CFO is a device-specific feature, i.e., it is related to the device’s RF circuit. (2) The CFO value is robust against interference from human movements as it only depends on RF signals’ frequency values rather than their amplitude values. Table 3 shows the results obtained by launching spoofing attacks on the nine BLE devices by the four attacker platforms. We observe that spoofing attacks can be robustly detected using the CFO inspection in most cases. Moreover, Table 4 shows the performance of CFO inspection when using different copies of the same BLE device as the benign BLE device and the attack device. We observe that even though the different copies have the same Bluetooth chip from the same manufacturer with the same version, the copy used as the BLE device can be readily differentiated from the copy used as the attacker.

In a few cases, BlueShield has relatively high FN values (i.e., higher than 15%). Our further analysis shows that such high FN values are mainly caused by the limited CFO reso-

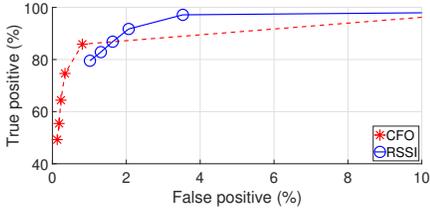


Figure 12: Receiver operating curve generated by changing the inspection threshold (Device ID = 9, $T_o = 0.3$ s.).

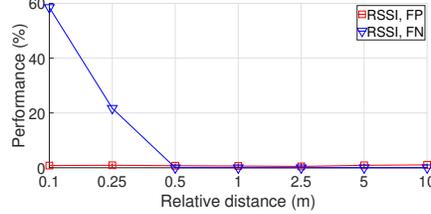


Figure 13: Detection performance vs. distance between BLE device and attacker (Device ID = 9, $\tau_r = 5$, $T_o = 0.3$ s.).

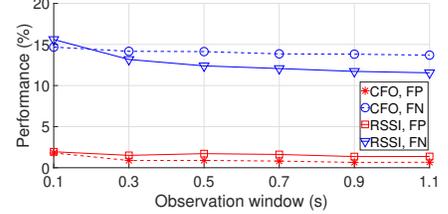


Figure 14: Detection performance vs. the duration of observation window (Device ID = 9, $\tau_c = 3$, $\tau_r = 5$).

Table 5: Effectiveness of the RSSI inspection in detecting packets transmitted from different locations.

Device ID	Night-time (8 pm-8 am)		Daytime (8 am-8 pm)		Average	
	FP	FN	FP	FN	FP	FN
1	1.25	0.00	0.69	9.44	0.97	5.84
2	3.00	5.82	4.18	33.62	3.59	21.15
3	0.11	4.19	2.75	4.18	1.43	5.22
4	0.57	3.36	2.06	37.07	1.32	22.94
5	1.68	0.46	0.84	2.12	1.26	1.60
6	1.22	0.41	2.05	2.32	1.64	1.46
7	1.34	0.02	1.02	5.86	1.18	3.00
8	0.40	2.88	1.29	0.05	0.85	1.55
9	1.10	5.92	1.85	20.47	1.48	15.73
Average	1.19	2.56	1.86	12.79	1.52	8.72

lution captured by Ubertooth One (the platform we used as the collector). More specifically, CFO in Ubertooth One is estimated with a resolution of 5.2KHz (p. 42 in [23]). This implies that any difference below 5.2KHz between two BLE devices cannot be detected by Ubertooth One. As such, a customized collector with a finer resolution will significantly enhance the detection performance. Nevertheless, even in these cases, the RSSI inspection alleviates such limitations, and the BlueShield’s overall FN rate is significantly lower than the CFO-only inspection.

RSSI Feature. We evaluate the effectiveness of RSSI inspection by analyzing the RSSI values of the attacker’s advertising packets from different locations. Figure 13 illustrates that spoofing attacks can be readily detected with low FN using the RSSI inspection when the attacker and the BLE device are placed at more than 0.25m. Further, we evaluate the RSSI inspection with (during daytime) and without (during night-time) significant interference from other wireless devices and human movements. In Table 5, we observe that while there is no significant impact of interference on the FP, the FN increases notably with the increase in interference. For instance, the FNs in detecting spoofing attacks on BLE devices with $ID = 2$ and $ID = 4$ are more than 30% during the daytime. This is because these BLE devices transmit advertising packets with lower signal power than other BLE devices, and the high channel interference conceals small differences in RSSI values of these BLE devices and attackers. We note

that for these cases, the CFO inspection naturally complements the RSSI inspection in effectively bringing down the BlueShield’s overall FN as shown in Table 2.

6.4 Responsiveness

The responsiveness of BlueShield is measured by the duration of the observation window T_o . Figure 14 illustrates that with increasing T_o , FP values of CFO and RSSI inspection do not change significantly, but their FN values decrease. Recall that BlueShield is implemented to report a spoofing attack by inspecting CFO and RSSI values of N_o advertising packets in the observation window T_o (Figure 7). As such, the specific values of N_o and T_o can be determined based on the required detection performance. We also highlight that since different BLE devices have different advertising periods, if we set N_o to a fixed value for all BLE devices, then T_o is different for different BLE devices, and vice versa.

For the results presented in Table 2, we configure BlueShield to have the same N_o but different T_o for different BLE devices. In the worst case, the observation window is limited to 6 s because of the requirement to achieve good detection performance for the BLE device ($ID = 4$) with the advertising period of 2 s. We point out that when BlueShield is deployed in real-world usage scenarios to monitor BLE devices with lower advertising periods, it can be optimized to have shorter observation windows. Overall, the fast responsiveness of BlueShield enables it to effectively detect the presence of the attacker before it can potentially spoof the user device. Such in advance alarms also enable BlueShield to quickly take other necessary actions (e.g., notify the network administrator) to *prevent* any harm to users.

6.5 Monitoring of Multiple BLE Devices

The number of BLE devices which BlueShield can monitor at the same time relies on the computation and communication capabilities of collectors and the monitor, and the channel interference in the monitored environment. We observe that with an increase in the number of monitored BLE devices, while the computational overhead at the monitor increases linearly, the computational burden at collectors does not change.

This is because collectors first need to record all packets on the three advertising channels and forward them to the monitor. The monitor is then responsible for classifying advertising packets belonging to different BLE devices and detecting spoofing attacks for each monitored BLE device. Our experiments also validate that BlueShield can monitor at least 30 BLE devices at the same time without any degradation in the detection performance corresponding to the monitored BLE devices. More detailed results are shown in Appendix D.

7 Discussion

Physical Features. BlueShield exploits the physical features of BLE devices' RF signals for device characterization. While some of these features (e.g., RSSI) have been previously employed to detect spoofing attacks in other wireless networks (e.g., Wi-Fi and ZigBee [10, 16, 35]), BlueShield augments three novel traits to enhance their effectiveness: (1) BlueShield employs a randomized channel switching mechanism that reflects the moving target defense and ensures a robust detection of spoofing attacks launched by an SDR-enabled attacker that can mimic the selected physical features. (2) BlueShield leverages cyber features (i.e., characteristics of the BLE protocol) to trigger the RSSI and CFO inspection at appropriate time instants. Such selective inspection allows BlueShield to significantly reduce false alarms compared to prior schemes. (3) Different from prior research which requires customized hardware to provide high-accuracy values of these features, BlueShield can be implemented using low-cost, off-the-shelf platforms without high-resolution and high-consistency RSSI and CFO values.

Deployment Considerations. The monitoring infrastructure required for implementing BlueShield (i.e., the collectors and the monitor) can be conveniently deployed on edge devices such as Wi-Fi/Bluetooth routers [9] which are widely utilized in indoor environments. These edge devices offer the natural choice to deploy BlueShield in a non-intrusive and economical way as they may already be equipped with BLE transceivers. For profiling a given BLE device, BlueShield only needs to make a one-time effort and collect a few advertising packets for determining the BLE device's relevant characteristics. Hence, the time needed to execute the profiling phase and add a new BLE device to the monitored environment is short (within a few seconds). Also, the one-time profiling process conveniently supports dynamic addition/removal of BLE devices to/from a BLE network monitored by BlueShield, without affecting other BLE devices in the network. In other words, there's no need to re-profile the monitored BLE devices. Besides, since the protocol characteristics of a BLE device are location-independent, they can also be securely profiled at an isolated location before deployment in the production environment.

8 Related work

Spoofing attacks against traditional wired and wireless networks have been widely studied in the existing literature which broadly includes GPS spoofing [42], DNS spoofing [38] and ARP spoofing [45]. In the context of BLE networks, the usage of encryption and authentication mechanisms provided in the BLE specification largely depends on the application-specific requirements determined by BLE device manufacturers. As such, the BLE devices which do not employ the recommended security mechanisms are vulnerable to a variety of attacks [3, 8, 13, 34] which can be trivially launched against their users. Recent studies [2, 49] have also revealed the design and implementation flaws of the BLE stack, which can be easily exploited by attackers to perform spoofing attacks against the BLE devices and their user devices which employ the specified authentication mechanisms. The detrimental implications of spoofing attacks on user devices have also been partly identified in prior research [26, 31].

To defend against spoofing attacks, the existing schemes largely rely on modifying the BLE protocol or updating the firmware/hardware of BLE devices [21, 37], on a vulnerability-by-vulnerability basis. Unfortunately, these approaches are impractical for wide adoption and deployment in the real world, especially for the millions of legacy BLE devices – many of which do not even support firmware updates – produced by many device vendors. One prior approach towards an off-the-shelf solution for protecting the privacy of BLE devices is BLE-Guardian [20], which mainly broadcasts jamming signals to corrupt the BLE device's advertising packets that contain privacy-sensitive information. Unfortunately, BLE-Guardian cannot defend against spoofing attacks. Instead, BlueShield is a practical, device-agnostic spoofing detection framework, which protects user devices against spoofing attacks without any interference or modification to the conventional BLE devices and user devices.

9 Conclusion

In this paper, we propose BlueShield, an out-of-the-box defense that provides a device-agnostic, legacy-friendly monitoring framework for detecting spoofing attacks in BLE networks. We demonstrate that BlueShield is robust against an advanced attacker with the ability to spoof the monitored features of a BLE device. BlueShield can be implemented using low-cost, off-the-shelf components; and its operation remains transparent to the communications between the user device and the BLE device. Our evaluation results illustrate that BlueShield can effectively and robustly detect spoofing attacks with very low false positive and false negative rates.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by ONR under Grant N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

- [1] Daniele Antonioli, Nils Tippenhauer, and Kasper Bonne Rasmussen. Key negotiation downgrade attacks on bluetooth and bluetooth low energy. *ACM Trans. Inf. Syst. Secur.*, 0(ja).
- [2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of Bluetooth BR/EDR. In *28th USENIX Security Symposium*, pages 1047–1061, August 2019.
- [3] Vaibhav Bedi. Exploiting BLE smart bulb security using BtleJuice: A step-by-step guide. <https://blog.attify.com/btlejuice-mitm-attack-smart-bulb/>, 2018. Accessed: August 1, 2019.
- [4] Bluetooth SIG. Bluetooth Market Update. <https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/>, 2019. Accessed: August 1, 2019.
- [5] Bluetooth SIG. Core Specifications 5.1. <https://www.bluetooth.com/specifications/bluetooth-core-specification>, 2019. Accessed: August 1, 2019.
- [6] Bluetooth SIG. Smart industry. <https://www.bluetooth.com/markets/smart-industry>, 2019. Accessed: August 1, 2019.
- [7] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless device identification with radio-metric signatures. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, pages 116–127, 2008.
- [8] Victor Casares. Mimo baby hack. https://medium.com/@victor_14768/mimo-baby-hack-ac7fa0ae3bfb, 2018. Accessed: August 1, 2019.
- [9] Cassia Networks. Cassia hub: The Bluetooth router. <https://www.cassianetworks.com/blog/cassia-hub-bluetooth-router/>, 2019. Accessed: August 1, 2019.
- [10] Yingying Chen, Jie Yang, Wade Trappe, and Richard P. Martin. Detecting and localizing identity-based attacks in wireless and sensor networks. *IEEE Transactions on Vehicular Technology*, 59(5):2418–2434, 2010.
- [11] CSR. CSR 4.0 Bluetooth USB adapter. https://www.amazon.com/Bluetooth-Adapter-Songway-Computer-Keyboard/dp/B07KWVXBKZ/ref=sr_1_46?keywords=bluetooth+adapter+car+4.0&qid=1563227361&s=electronics&sr=1-46. Accessed: August 1, 2019.
- [12] Cypress. CYW920735Q60EVB-01 Evaluation Kit. <https://www.cypress.com/documentation/development-kitsboards/cyw920735q60evb-01-evaluation-kit>. Accessed: August 1, 2019.
- [13] Aavek K. Das, Parth H. Pathak, Chen-Nee Chuah, and Prasant Mohapatra. Uncovering privacy leakage in BLE network traffic of wearable fitness trackers. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 99–104, 2016.
- [14] National Vulnerability Database. CVE-2020-9970. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9770>. Accessed: June 26, 2020.
- [15] Dell. Dell Latitude 5480. <https://www.dell.com/en-us/work/shop/dell-laptops-and-notebooks/latitude-5480/spd/latitude-14-5480-laptop>. Accessed: August 1, 2019.
- [16] Murat Demirbas and Youngwhan Song. An RSSI-based scheme for sybil attack detection in wireless sensor networks. In *Proceedings of the International Symposium on World of Wireless, Mobile and Multimedia Networks*, pages 564–570, 2006.
- [17] Developex. BLE in smart home devices. <https://developex.com/blog/ble-in-smart-home-devices/>, 2017. Accessed: Aug 1, 2019.
- [18] Medical Device and Diagnostic Industry. 3 reasons bluetooth is perfect in healthcare settings. <https://www.mddionline.com/3-reasons-bluetooth-perfect-healthcare-settings>, 2019. Accessed: August 1, 2019.
- [19] Ramsey Faragher and Robert Harle. Location fingerprinting with Bluetooth low energy beacons. *IEEE Journal on Selected Areas in Communications*, 33(11):2418–2428, 2015.
- [20] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting privacy of BLE device users. In *25th USENIX Security Symposium*, pages 1205–1221, 2016.

- [21] Avinatan Hassidim, Yossi Matias, Moti Yung, and Alon Ziv. Ephemeral identifiers: Mitigating tracking & spoofing threats to BLE beacons. <https://developers.google.com/beacons/eddystone-aidpreprint.pdf>, 2016. Accessed: August 1, 2019.
- [22] Software Testing Help. 18 most popular iot devices in 2020. <https://www.softwaretestinghelp.com/iot-devices/>. Accessed: August 1, 2019.
- [23] Texas Instruments. CC2400 data sheet. <http://www.ti.com/lit/ds/symlink/cc2400.pdf>, 2006. Accessed: August 1, 2019.
- [24] Zhu Jianyong, Luo Haiyong, Chen Zili, and Li Zhaohui. RSSI based Bluetooth low energy indoor positioning. In *International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 526–533, 2014.
- [25] Keystudio. Ks0255 keystudio bluetooth 4.0 expansion shield. https://wiki.keystudio.com/Ks0255_keystudio_Bluetooth_4.0_Expansion_Shield, 2019. Accessed: August 1, 2019.
- [26] Constantinos Koliass, Lucas Copi, Fengwei Zhang, and Angelos Stavrou. Breaking BLE beacons for fun but mostly profit. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, pages 4:1–4:6, 2017.
- [27] Kontakt.io. Beacons in healthcare. <https://goto.kontakt.io/beacons-in-healthcare>, 2019. Accessed: August 1, 2019.
- [28] Angela Lonsetta, Peter Cope, Joseph Campbell, Bassam Mohd, and Thair Hayajneh. Security vulnerabilities in Bluetooth technology as used in IoT. *Journal of Sensor and Actuator Networks*, 7(3):28, 2018.
- [29] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.
- [30] Tal Melamed. BLE Application Hacking. https://www.owasp.org/images/archive/6/6f/20170811005623%21OWASP2017_HackingBLEApplications_TalMelamed.pdf, 2017. Accessed: August 1, 2019.
- [31] Tal Melamed. An active man-in-the-middle attack on bluetooth smart devices. *Safety and Security Studies (2018)*, 15, 2018.
- [32] Rajani Muraleedharan and Lisa Ann Osadciw. Jamming attack detection and countermeasures in wireless sensor network using ant system. In *Wireless Sensing and Processing*, volume 6248, page 62480G. International Society for Optics and Photonics, 2006.
- [33] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–14, 2014.
- [34] Mike Ryan. Bluetooth: With low energy comes low security. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, pages 1–7, 2013.
- [35] Yong Sheng, Keren Tan, Guanling Chen, David Kotz, and Andrew Campbell. Detecting 802.11 MAC layer spoofing using received signal strength. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1768–1776, 2008.
- [36] GARY SIMS. Bluetooth mesh positioned to provide de-facto protocol for smart homes. <https://www.androidauthority.com/bluetooth-mesh-for-smart-homes-940886/>, 2019. Accessed: August 1, 2019.
- [37] Pallavi Sivakumaran and Jorge Blasco. A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape. In *USENIX Security Symposium*, pages 1–18, 2019.
- [38] U Steinhoff, A Wiesmaier, and R Araújo. The state of the art in dns spoofing. In *Proc. 4th Intl. Conf. Applied Cryptography and Network Security (ACNS)*, 2006.
- [39] Weiping Sun, Jeongyeup Paek, and Sunghyun Choi. CV-track: Leveraging carrier frequency offset variation for BLE signal detection. In *Proceedings of the 4th ACM Workshop on Hot Topics in Wireless (HotWireless)*, pages 1–5, 2017.
- [40] John Teel. How to develop a sellable bluetooth low-energy (BLE) product. <https://makezine.com/2016/08/01/develop-sellable-bluetooth-low-energy-ble-product/>. Accessed: April 6, 2020.
- [41] Ubertooth Developers. Ubertooth One. <https://github.com/greatscottgadgets/ubertooth/wiki>, 2019. Accessed: August 1, 2019.
- [42] E. Vattapparamban, İ. Güvenç, A. İ. Yurekli, K. Akkaya, and S. Uluğaç. Drones for smart cities: Issues in cybersecurity, privacy, and public safety. In *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 216–221, 2016.

- [43] Tien Dang Vo-Huu, Triet Dang Vo-Huu, and Guevara Noubir. Fingerprinting Wi-Fi devices using software defined radios. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 3–14, 2016.
- [44] Voicebot. Smart speaker consumer adoption report 2018. <https://voicebot.ai/2018/04/02/smart-speaker-owners-use-voice-assistants-nearly-3-times-per-day/>. Accessed: August 1, 2019.
- [45] Sean Whalen. An introduction to arp spoofing. *Node99 [Online Document]*, April, 2001.
- [46] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Antonio Tian, Dave (Jing) Bianchi, Mathias Payer, and Dongyan Xu. Blesa: Spoofing attacks against recon-nections in bluetooth low energy. In *Proceeding of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)*, 2020.
- [47] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious Bluetooth peripherals. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, pages 1–15, 2019.
- [48] Wenyuan Xu, Ke Ma, Wade Trappe, and Yanyong Zhang. Jamming sensor networks: attack and defense strategies. *IEEE network*, 20(3):41–47, 2006.
- [49] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. On the (in)security of bluetooth low energy one-way secure connections only mode, 2019.
- [50] Yuan Zhuang, Jun Yang, You Li, Longning Qi, and Naser El-Sheimy. Smartphone-based indoor localization with Bluetooth low energy beacons. *Sensors*, 16(5):596, 2016.

A RSSI Inspection

Inspection mechanism. To detect anomalies in RSSI values under the presence of strong signal reflections in BLE networks, we employ the two-component Gaussian mixture model. This is because the distribution of RSSI values in noise-rich environments can be modeled using two normal distributions [35]. The probability distribution function of RSSI values can be represented by

$$f_r(y_i) = w \cdot \frac{1}{\sigma_1 \sqrt{2\pi}} \cdot e^{-\frac{(y_i - \mu_1)^2}{2\sigma_1^2}} + (1 - w) \cdot \frac{1}{\sigma_2 \sqrt{2\pi}} \cdot e^{-\frac{(y_i - \mu_2)^2}{2\sigma_2^2}}, \quad (1)$$

where μ_1 and μ_2 represent means of the two components, σ_1 and σ_2 represent standard deviations of the two components, w represents a weight parameter, and y_i denotes an RSSI sample. In BlueShield, the monitor utilizes N_l RSSI values of advertising packets in the lookback window to learn values of μ_1 , μ_2 , σ_1 , σ_2 and w using a conventional expectation–maximization (EM) algorithm [29]. Then, the monitor computes the negative log-likelihood that RSSI values (y_i , $\forall i \in [1, N_o]$) of advertising packets in the observation window come from the model given by equation (1), i.e.,

$$L_r = \frac{1}{N_o} \sum_{i=1}^{N_o} -\log f_r(y_i), \quad (2)$$

Finally, the monitor detects an anomaly if the negative log-likelihood is larger than an RSSI inspection threshold denoted by τ_r , i.e., $L_r > \tau_r$. Here, τ_r is a design parameter in BlueShield which is further discussed in Section 6.

B BlueShield App

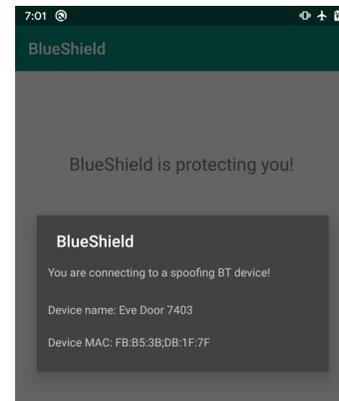


Figure 15: Sample of a notification of a spoofing attack shown in BlueShield’s Android app.

We have developed an Android app which communicates with BlueShield’s monitor through a secure HTTPS connection. If the user registers to BlueShield’s notification service, the user can receive the notification about the spoofing attack with relevant information, such as the targeted BLE device’s name and MAC address (Figure 15).

C Analysis of INT inspection

Here, by theoretically analyzing the INT inspection from the perspective of the lower bound of INT, we illustrate that the INT inspection mechanism encounters negligible false negative in detecting spoofing attacks.

In BlueShield, the lower bound of INT, T_{lb} , is calculated such that $T_{lb} \in [T_p - \Delta, T_p]$, where T_p is the advertising period

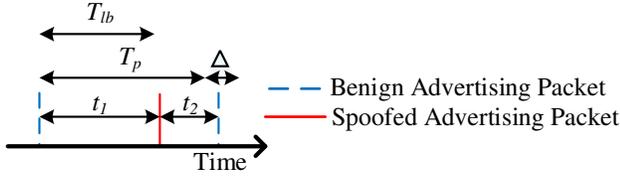


Figure 16: Illustration for the INT inspection analysis.

of the device and $\Delta = 10\text{ms}$. In the spoofing attack, when the device and the attacker broadcast advertising packets simultaneously, the monitor can observe a spoofed advertising packet between two consecutive benign advertising packets (Figure 16). We note that the original INT is divided into two observed INT values, t_1 and t_2 . This spoofing attack can be detected if at least one of t_1 and t_2 is smaller than T_{lb} . Clearly, when $t_1 < T_{lb}$, the attack can be detected. When $t_1 \geq T_{lb}$, the INT inspection can miss the attack if $t_2 \geq T_{lb}$. Such a false negative can be produced under the following condition.

$$\begin{aligned}
 & t_1 + t_2 \leq T_p + \Delta \\
 \implies & 2T_{lb} \leq T_p + \Delta \quad (\text{since } T_{lb} \leq t_1 \text{ and } T_{lb} \leq t_2) \\
 \implies & 2T_p - 2\Delta \leq T_p + \Delta \quad (\text{since } T_p - \Delta \leq T_{lb}) \\
 \implies & T_p \leq 3\Delta
 \end{aligned}$$

The above expression implies that when $\Delta = 10\text{ms}$, the attacker can bypass the INT inspection if $T_p \leq 30\text{ms}$. We highlight that as per the BLE specification (p.1322 in [5]), $T_p \geq 20\text{ms}$. In our experiments (Table 2), since the minimum value of T_p among the nine test devices is 100ms, we do not observe such a false negative.

However, the packet loss rate in the BLE network might affect the false negative of the INT inspection in BlueShield. Specifically, the INT inspection may fail to detect an anomaly in the advertising interval when the collector does not receive the benign advertising packet due to extensive interference, but it receives the spoofed advertising packet from the attacker. In this scenario, when the packet loss rate of the benign advertising packet is denoted by ρ , the probability for the attacker to transmit n spoofed advertising packets without being detected by the INT inspection is ρ^n . Hence, the resulting FN

Table 6: Comparison of BlueShield’s performance when monitoring 1 BLE device and 9 BLE devices.

Device ID	Packet Loss (%)		Packet Delay (ms)	
	1 device	9 devices	1 device	9 devices
1	7.50	9.35	11.43	10.42
2	20.46	15.79	11.95	11.25
3	6.00	5.79	11.38	10.71
4	6.35	8.81	10.32	11.16
5	15.12	14.03	11.11	10.66
6	8.07	8.93	6.86	8.57
7	4.57	6.29	10.15	11.51
8	8.70	12.73	10.73	11.29
9	8.00	10.15	11.10	10.69
Average	9.42	10.21	10.56	10.70

value is negligible. For instance, when $\rho = 10\%$ (which is the typical packet loss rate observed in our experimental setup as shown in Table 6), the probability to transmit $n = 6$ spoofed advertising packets (in two advertising events) without being detected is 0.0001%. We highlight that six is an extremely small number of packets for the spoofing attack to succeed in a real-world environment.

D Impact of Multiple Device Monitoring

We evaluate the effectiveness of BlueShield in monitoring multiple BLE devices in terms of two metrics: (1) *packet loss rate* defined as the ratio of the number of packets received by collectors and the number of packets transmitted by the BLE device, and (2) *packet delay* defined as the time required by the monitor to retrieve a packet from a collector and inspect its features. As can be seen from Table 6, our results indicate that there is no significant difference in the average packet loss rate (9.42% vs. 10.21%) and the average packet delay (10.56ms vs. 10.70ms) between monitoring one BLE device and 9 BLE devices, respectively. Further, recall that in our experiments, BlueShield readily discovered 30 BLE/Bluetooth equipped devices in our office environment. This indicates that BlueShield can effectively monitor at least 30 BLE devices, which covers most BLE usage scenarios.

Dark Firmware: A Systematic Approach to Exploring Application Security Risks in the Presence of Untrusted Firmware

Duha Ibdah, Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, Hafiz Malik
University of Michigan, Dearborn, USA
{dhibdah, nlachtar, abdkhail, bacha, hafiz}@umich.edu

Abstract

Compromising lower levels of the computing stack is attractive to attackers since malware that resides in layers that span firmware and hardware are notoriously difficult to detect and remove. This trend raises concerns about the security of the system components that we have grown accustomed to trusting, especially as the number of supply chain attacks continues to rise. In this work, we explore the risks associated with application security in the presence of untrusted firmware. We present a novel firmware attack that leverages system management cycles to covertly collect data from the application layer. We show that system interrupts that are used for managing the platform, can be leveraged to extract sensitive application data from outgoing requests even when the HTTPS protocol is used. We evaluate the robustness of our attack under diverse and stressful application usage conditions running on Ubuntu 18.04 and Android 8.1 operating systems. We conduct a proof-of-concept implementation of the attack using firmware configured to run with the aforementioned OSs and a mix of popular applications without disrupting the normal functionality of the system. Finally, we discuss a possible countermeasure that can be used to defend against firmware attacks.

1 Introduction

Traditionally, firmware has been perceived as the invisible software that breathes life into the digital world around us at the stroke of a button. In addition to firmware's role of silently configuring and initializing platform resources, it serves the purpose of maintaining the health of the underlying hardware through periodic management cycles.

Although firmware is often regarded as a decoupled entity from the application layer, the aforementioned management cycles could be re-purposed to scrape system memory and therefore undermine the confidentiality guarantees systems provide for user data. As a result, it is critical to systematically evaluate the security of systems when faced with adversaries

that can harness firmware to maliciously scrape memory for sensitive user data. This is especially important as the number of supply chain attacks that can result in malicious firmware being installed on systems continues to rise [1–3].

A significant body of work has explored various attacks aimed at harvesting sensitive data from system memory [4–13]. Work by Hizver et al. [4] demonstrated the use of introspection attacks through hypervisors to extract credit card data from Point of Sale (PoS) systems running as virtual machines. Other work examined memory scraping techniques that involve physical access [6–13]. For example, Halderman et al. [6] explored how memory modules could be extracted from a machine and then scanned to recover cryptographic keys. Similar work investigated the re-purposing of different hardware ports to collect memory dumps from smartphones once a device has been stolen [8]. Other work [9–13], considered the use of malicious peripherals to collect memory dumps through DMA attacks.

Unlike prior work, we propose a novel firmware-based attack that covertly scrapes memory for sensitive user data. Our attack doesn't require the use of special hardware or physical access. It is not limited to environments that employ virtual machines and can be applied to most computing systems. However, despite such advantages, the low-level nature of this subsystem makes it difficult to gain insight into the application layer and how its data is managed. This lack of insight introduces a layer of complexity for systematically collecting sensitive user data through firmware. Another challenge relates to the limited execution cycles firmware is allocated during runtime. Reclaiming compute resources beyond the intended time slice can cause the system to malfunction and applications to become unresponsive. This is especially true for mobile environments where app responsiveness and energy efficiency are treated as first order constraints for consumers.

In this paper, we explore the possibility of leveraging system management cycles to collect user passwords from outgoing HTTP requests. We show that periodic system interrupts that are used for managing platform events, could be harnessed to reliably and efficiently extract sensitive user data

through the use of malicious firmware. We show that this is possible even when applications employ the secure HTTPS protocol. We evaluate the effectiveness of this approach across desktop and mobile systems by extensively testing popular web services and mobile apps, such as Instagram, Facebook, and LinkedIn running on different hardware configurations. We characterize the robustness of our approach under stressful app usage conditions while considering a diverse set of applications from six different categories. We build a proof of concept for our proposed attack using real system firmware that is configured to run with Ubuntu 18.04 Linux and Oreo 8.1 Android operating systems combined with a mix of popular web services and apps. We devise a mechanism that achieves low overhead across both x86-64 and ARMv8 architectures. We accomplish this by limiting memory scans to the user accessible dirty pages in memory that obviates the need for parsing the entire memory subsystem. Finally, we discuss a possible hardware-based countermeasure that can be used to defend against such attacks.

Overall, this paper makes the following contributions:

- Presents a novel attack that leverages platform management cycles to collect sensitive data from HTTP requests even when HTTPS is used.
- Conducts a proof-of-concept implementation of the attack using real firmware configured to run with Ubuntu Linux and Android Oreo operating systems.
- Characterizes the robustness of the proposed approach while running a diverse set of popular web services and mobile apps under different platform configurations.
- Demonstrates how firmware can efficiently leverage page tables to covertly extract sensitive information without disrupting the normal functionality of desktop and mobile systems, and discusses a possible countermeasure that could be employed to defend against such attacks.

The rest of this paper is organized as follows: Section 2 provides background and motivation information for our attack based on experimental data. Section 3 illustrates the threat model. Section 4 details the design and algorithms for the proposed attack. Section 5 presents an experimental evaluation. Section 7 details related work; and Section 8 concludes.

2 Background and Motivation

2.1 System Firmware

System firmware is an essential component that abstracts away hardware details from the OS as a way of providing a common OS agnostic view across multiple platforms. It can be divided into two phases: boot time and runtime. The boot time phase is responsible for initializing the platform’s hardware resources during in preparation for launching the

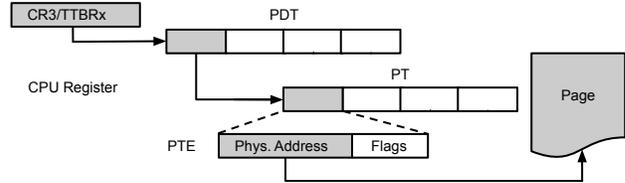


Figure 1: A high level overview of the address translation process from virtual to physical memory.

OS. Although the main responsibility of system firmware entails discovering and initializing the hardware resources, it also plays a role in providing runtime services to the OS. Since the OS doesn’t have enough insight about the details of a given platform, it relies on firmware to perform certain actions on its behalf. Such actions include the gathering and handling of machine state information in response to hardware events, as well as housekeeping tasks that include power management [14, 15] and system authentication [16].

Transfer of control from the OS layer into system firmware is achieved through a special hardware interrupt. This mechanism enables firmware to conduct platform management tasks transparently from the OS. Therefore, before passing control to the OS, it is incumbent upon firmware to configure the underlying hardware to generate a set of manageability interrupts. This mechanism prompts the interrupted core to invoke firmware execution and put the processor into a special mode that we will refer to as management mode. This mode grants firmware access to system resources. In addition, this mode shadows resources away from the OS and application layers. As such, any transactions issued by firmware would not be visible to upper layers.

2.2 Virtual Address Space

Virtual address space is a fundamental component that provides isolation between processes while presenting each process with the illusion that it has access to the entire address space. The OS accomplishes this through paging, a mechanism that maps process virtual addresses into physical ones. Virtual address transactions issued by a given process result in the kernel walking a series of translation tables that it uses to determine the target physical address of the mapped page. The physical addresses that mark the start of such tables are tracked through dedicated CPU registers. x86 systems make use of a single register (CR3) to maintain both user and kernel space addresses, while ARM uses separate registers for mapping user and kernel space addresses (TTBR0 and TTBR1).

Translation tables are organized in a directory like structure often consisting of a page directory table (PDT) followed by page table entries (PTE). However, this directory structure can consist of multiple levels that include intermediate directories. The page table entries serve an important role since they hold

the physical addresses of the pages that represent their virtual address counterparts. In addition, each PTE contains different flags that describe the properties of the page it's associated with. Typical flags include bits that are used to indicate the presence of a page in memory, whether a page is dirty, and if a page is accessible from user space. A high level overview of the address translation process is illustrated in Figure 1.

2.3 The Case for Firmware-based Attacks

According to Shane Wall, the director of HP Labs [17], compromises at the firmware level are attractive to attackers since malicious firmware remains persistent regardless of OS re-installations and storage reformatting, often necessitating a hardware replacement. Since our attack relies on the presence of malicious firmware on the system, we discuss possible ways this could be accomplished.

Pushing malicious firmware to systems can be performed through the software supply chain process by compromising live update utilities that are shipped with systems to accommodate future updates. Such an attack was demonstrated by Operation Shadow Hammer that was discovered in 2019 by Kaspersky [3]. Similar attacks can also be performed after deployment by exploiting vulnerabilities in the firmware update process administered by the OS. For instance, prior work explored how vulnerabilities could be leveraged for injecting malicious firmware including the use of remote updates that can be performed from user space while bypassing existing safety measures, such as secure boot [18–24].

Unfortunately, firmware related vulnerabilities show no sign of abating as this component continues to evolve to promote features such as remote access and over the internet updates. Such features could be harnessed by cybercriminals to enable sensitive data collection across a multitude of platforms that span mobile devices, computer systems, and network infrastructure. Figure 2 summarizes this trend by showing the number of firmware related Common Vulnerabilities and Exposures (CVE) reported by the national vulnerability database [25]. On average, 437 firmware CVEs are reported every year with 39 of such CVEs detailing potential exposure to malicious firmware updates. This trend highlights the increased susceptibility of systems to untrusted firmware and the importance of understanding the impact of such risks.

2.4 The Case for HTTP Attacks

An indispensable technology that pre-enables the delivery of cloud-based services is the Hypertext Transfer Protocol (HTTP). HTTP supports two primary mechanisms for issuing requests: GET and POST. Data transferred through the GET method is accomplished by embedding the information directly within the Uniform Resource Locator (URL). For example, `http://www.mypage.com/form.php?name1=value1&name2=value2` can be used to send input `value1` and `value2`

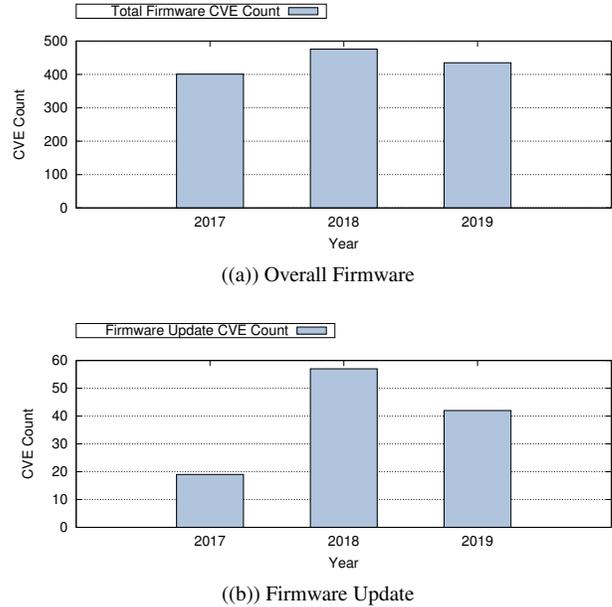


Figure 2: Number of Common Vulnerabilities and Exposures (CVE) per year for (a) all firmware vulnerabilities and (b) vulnerabilities only affecting firmware update.

for form fields `name1` and `name2` to `www.mypage.com`. On the other hand, transferring data through POST is achieved through a body section that is appended to the header portion of a given HTTP request.

Service requests initiated by users are typically performed through web browsers and mobile apps using POST. Such transactions often involve sending login information over HTTP to allow service providers to authenticate their users. Although HTTP is employed for sending sensitive information such as login credentials, the protocol itself is not designed to provide confidentiality guarantees. Instead, confidentiality guarantees for application data sent over HTTP is accomplished through the Transport Layer Security (TLS) which is implemented between the application and transport layers. TLS is designed to make data exchanged between a client (browser/app) and a server (service provider) cryptographically secure (HTTPS). Unfortunately, this layered approach for securing data is susceptible to adversaries that can intercept sensitive HTTP data before it is encrypted by the underlying TLS layer.

In order to evaluate the feasibility of recovering sensitive data, we characterized two popular services, namely Facebook and Instagram. In the first experiment, we focused on collecting Facebook usernames and passwords after logging into the service through a Chrome web browser that ran on an Ubuntu 18.04 system configured with 4 CPU cores and 8 GB of memory. Upon logging into Facebook, we collected a snapshot of the memory subsystem and analyzed its contents.

During this process, we discovered different occurrences of our Facebook login credentials in memory. For instance, we were able to locate the password embedded in the relevant form fields of the actual Facebook page that was rendered by the local browser. In addition, we found the HTTP request associated with sending our Facebook username and password information over the network. A sample of this HTTP request is shown in Figure 3(a). We observed that the body of the POST request collected from memory consisted of the email and pass parameters xyz@gmail.com and pass123, but presented in URL encoding instead.

The second experiment entailed using Instagram. However, this experiment focused on collecting the credentials for this service after using Instagram’s mobile app. We downloaded the Instagram app onto an Android Oreo 8.1 system that ran on a system configured with 4 CPU cores and 4GB of memory. Similar to the Facebook experiment, we collected a snapshot of the memory subsystem and analyzed the contents after we had logged into the app. We observed multiple occurrences of our Instagram password in formats that conform to a JavaScript Object Notation (JSON). In addition, we were able to locate the HTTP POST request that was used to send out the authentication information using the JSON format. This information is illustrated in Figure 3(b).

The aforementioned Facebook and Instagram experiments demonstrate the feasibility of extracting authentication information from HTTP requests that are issued through either web browsers or mobile apps. Similar experiments show that we were able to successfully collect usernames and passwords across other popular services such as Gmail, Twitter, Facebook Messenger, and LinkedIn. Such findings underscore the potential for maliciously leveraging system firmware to covertly parse HTTP headers and extract secret data.

3 Threat Model

This study assumes a threat model that is consistent with prior work on firmware attacks. More specifically, we assume the attacker has the ability to install malicious firmware onto a system. A large body of work has demonstrated the ability to inject malicious firmware into a system [18–24]. For example, [21] demonstrated how to update the firmware directly from Windows by exploiting vulnerabilities in the firmware update process. Furthermore, an attacker could compromise the supply chain of the system manufacturer and in turn leverage the manufacturer’s live update utility to push malicious firmware across a large number of systems as in the case of Shadow Hammer with ASUS systems [3] which led to 57,000 users having a backdoored version of the live update utility. An attacker could also use spear phishing techniques as was recently done with LoJax [26], a malicious application that runs code that infects the platform’s firmware.

```
Host: www.facebook.com
Method: POST
Path: /login/device-based/regular/login
Content-Type: application/x-www-form-urlencoded

jazoest=2697&lsd=AVrRLRxH&email=xyz%40gmail.com&pass=pass123&timezone=300&lgndim=eyJ3IjoyNTYwLCJoIjoxNDQwLCJhdYI6MjU2MCw
```

((a)) Facebook

```
Host: i.instagram.com
Method: POST
Path: /api/v1/accounts/login
Content-Type: application/json

{"phone_id": "2ecdfcef-30d8-4678-a771-424d2161f602", "username": "xyz@gmail.com", "device_id": "android-f74b7d8d404d1cbe", "password": "pass123", "login_attempt_coun
```

((b)) Instagram

Figure 3: Login patterns obtained from HTTP authentication requests for (a) Facebook web service and (b) Instagram Android app.

4 Attack Prototype

Our attack is designed to explore the security risks associated with modern firmware and their impact on computing environments that range from mobile devices to computer systems. To this end, we devise a mechanism that transparently undermines the confidentiality guarantees provided by the upper layers and devise a framework that can systematically collect authentication information from HTTP requests without impacting normal execution. We developed system firmware using the Universal Extensible Framework Interface (UEFI). An overview of our prototype’s execution flow is outlined in Figure 4. We use a daisy-chained multicore approach for parsing HTTP requests present in memory. This approach allows for efficient streamlining of the search process across multiple cores. It also minimizes the number of cycles each core is taken away from the user.

The execution flow starts by invoking the lowest numbered core (core 0) into firmware through a periodic management interrupt. The periodic interrupt is configured in hardware through our firmware during the platform initialization phase and prior to the OS taking ownership of the system. This is depicted as step 1 in Figure 4. During this phase, the core consumes a set of predefined rules that it uses for parsing HTTP requests present in memory. HTTP requests are located by parsing each page for a pattern that begins with a valid request-line of an HTTP header (starts with "POST" and ends with CR + LF). The body of the request is then searched for formats that use either key/value form fields or JSON. As a proof of concept, the parameters of the body are parsed against keywords such as username, email, and password.

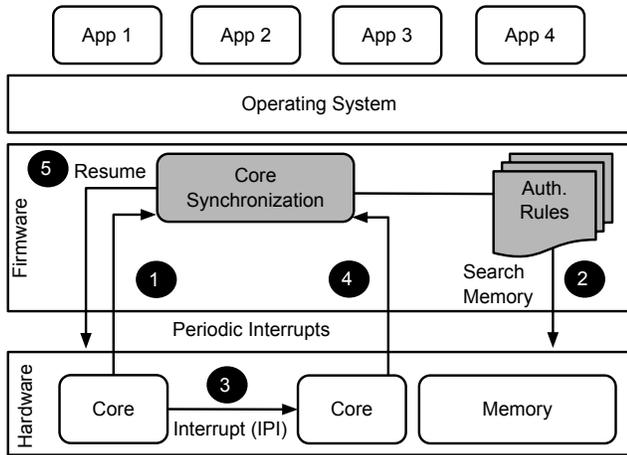


Figure 4: Main components of our firmware prototype for extracting user credentials from HTTP requests.

This is shown as step 2. Once step 2 is started, the core initiates its search for possible user credentials based on the rules it previously consumed. The core continues parsing memory until its time slice expires. When this occurs, the core halts the search and issues an inter-processor interrupt (IPI) to the next core (core 1) that will be responsible for resuming the search (step 3). However, before this newly invoked core (core 1) resumes the search, both cores (cores 0 and 1) undergo a synchronization phase to share information. This includes sharing the address of the last memory location that was parsed by the exiting core (core 0), a pointer to the last PTE that was referenced (both PTE and PDT information), as well as, a pointer to the next data structure that should be used for storing recovered credentials (step 4). Once the synchronization phase is complete, the previous core (core 0) is relinquished by firmware to resume normal execution while the newly invoked core (core 1) resumes the search (step 5). This process of having cores resuming memory searches followed by IPIs to the next available core, continues until the last core in the system is invoked into firmware. Once the last core completes its time slice, system firmware becomes inactive and awaits the next periodic management interrupt to occur. The execution flow from Figure 4 restarts again with step 1 once hardware issues the next periodic management interrupt.

A challenge with re-purposing manageability cycles for extracting user credentials relates to performance. Full memory scans require time to complete and varies as a function of memory capacity. This can lead to user applications to malfunction or become unresponsive. To reduce the performance impact on the overall system, our attack relies on walking the OS’s page tables and examining the page table entries before scanning any pages. This process is illustrated in Figure 5. Whenever the initial core (core 0) receives the very first man-

agement interrupt, it copies the physical address of the page tables from the CPU register and begins walking the structures. For every encountered PTE, the firmware examines the PTE flags to determine if the page should be parsed. The firmware only parses a given page if its dirty, user/supervisor, and write flags (D/U/W) are set. Otherwise, the page is skipped. This approach greatly reduces the number of manageability cycles used by firmware and makes the design dependent on the number of launched apps instead of the memory capacity of the system.

Furthermore, our design makes use of memory that is available through runtime services to save collected data. Unlike memory allocated through boot time services, the OS doesn’t reclaim memory regions that are reserved through runtime services, and therefore, remain available to firmware after the OS has taken ownership of the system. We utilize this memory for buffering recovered passwords from HTTP requests before they are saved to non-volatile memory. The saved data can then be ex-filtrated to a command-and-control (C&C) server through firmware’s own network stack during a reboot of the platform. To enable this, we include the UEFI network stack within our firmware and dispatch it during boot to enable communication with a C&C server.

Although our attack relies on reboots for ex-filtrating data, we discuss how our design could be extended to ex-filtrate data while the OS is running. This approach, however, requires additional support across firmware’s boot time and runtime phases. The aforementioned entails dispatching the network stack as a runtime module during boot and strictly consuming memory available through runtime services. This ensures that any resources that are consumed by the network stack are not reclaimed after control is relinquished to the OS. Sending data over the network while the OS is running requires an additional step. Specifically, firmware must invoke all CPU cores from the OS to avoid any contention over the network device. This can be achieved by having the master core that receives the management interrupt generate an IPI to the remaining cores in order to synchronize them within firmware. Data can be sent over the network once the cores have been synchronized, then relinquished back to the OS



Figure 5: Example of firmware-based attack selecting pages for parsing according to their page table entry (PTE) flags. Pages that do not have their dirty (D), user/supervisor (U), and write (W) flags set are skipped to improve performance.

after the data has been transmitted. This process can repeat as needed over a predefined period (e.g. once every 24 hours).

5 Evaluation

5.1 Experimental Framework

We conducted experiments using several applications across desktop and mobile platforms. To ensure test coverage of our solution under different application requirements, we used a diverse set of applications from six categories: social, communication, productivity, travel & local, health & fitness, and entertainment. The aforementioned categories and their corresponding applications are summarized in Table 1. In order to test our desktop system, we downloaded and installed commonly used applications for each category including productivity programs such as Office suite, Android Studio, and Eclipse that were configured to run on Linux. In addition, we launched common web services such as YouTube, Google Maps, and TripAdvisor through Google Chrome on the same platform. In the case of our mobile system, we downloaded several Android apps from Google Play store. Since computer systems are used in the context of productivity, we tested our desktop environment (Ubuntu) against more productivity applications compared to the Android apps used with our mobile platform (Android). We used a synthetic workload from [27] to serve as a stress application across both desktop and mobile platforms. Each stress application consisted of memory allocations that were used to increase the pressure on the memory subsystem and induce low memory conditions.

We used Ubuntu 18.04 LTS for running our linux-based desktop system and the Android Open Source Project (AOSP) 8.1 release for running Android. We installed Termux with BusyBox version 1.22.1 on Android in order to launch the stress apps from [27]. We used the QEMU 3.0.50 release for building our desktop and mobile platforms. This allowed us to test multiple hardware configurations as part of characterizing the robustness of our attack. In addition, we created a test harness using Python 3.6 that collected and analyzed memory snapshots through QEMU’s debug monitor at different intervals. These snapshots were used for analyzing the outgoing HTTP requests and constructing the username and password rules for different web-based services and Android apps. Examples of such patterns are shown in Figure 3. The test harness was also responsible for generating management interrupts to invoke firmware using different interrupt rates.

We developed system firmware using the Universal Extensible Framework Interface (UEFI) that we built from the Tianocore open source project. Finally, we used the gem5 simulator [28] to collect cycle accurate information and analyze the performance overhead of our attack. We modeled a 3.7GHz x86-64 processor and 2.1GHz ARMv8 processor with DDR4 memory. The parameters of the hardware configurations we modeled are summarized in Table 2.

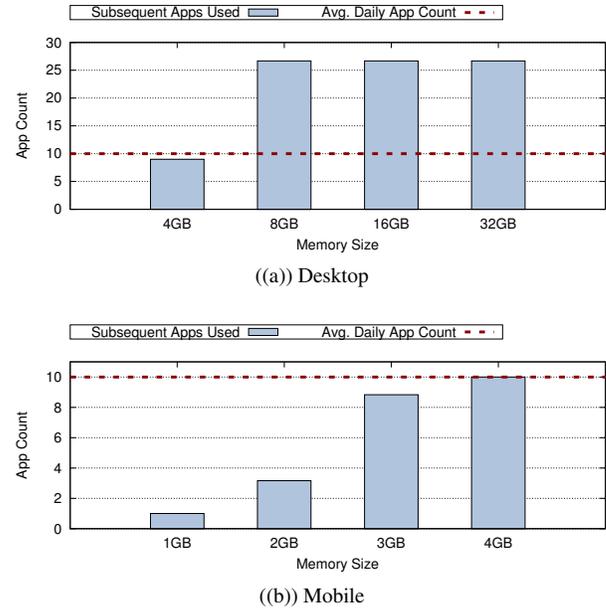


Figure 6: Summary of credential information persistence as a function of newly launched apps after user login.

5.2 Persistence vs. Application Usage

We first tested our firmware’s ability to extract authentication credentials at runtime from memory under different app usage conditions. To this end, we enabled our firmware with a set of rules based on HTTP requests, Facebook’s web service, and Instagram’s Android app issue for authenticating their users. We analyzed several memory dumps of the running services and determined a reliable set of rules that could be used. We then characterized the ability of our firmware to retrieve Facebook’s credentials on a desktop platform after launching 50 applications. We also conducted a similar experiment with Instagram after launching 20 apps on a mobile platform. The categories used in this experiment are listed in Table 1.

Characterization entailed constructing six app mixes according to Table 3 to simulate the persistence of login credentials under different app use cases. For each test instance, we logged into Facebook/Instagram (desktop/mobile), launched a different sequence of mixes, and then assessed the persistence of the login information after each app launched from a given mix while using our firmware. The experiments consisted of dividing 50 apps (desktop) and 20 apps (mobile) into six app mixes where each app mix consisted of all six categories that were run in a round robin fashion. The order of the apps launched in each category conformed to the same order listed in Table 1. Apps in each mix were used to generate system activity through actions, such as streaming videos, playing games, opening files, getting directions to an address, and reading emails for a duration of 30 seconds.

Figure 6 shows the results of the persistence experiment across our desktop and mobile platforms under different mem-

Category	Desktop Applications (Ubuntu)	Mobile Applications (Android)
Social	<i>Corebird (Twitter), Reddit, LinkedIn, Pinterest, Ramme (Instagram), Tumblr, Nextdoor, Wattpad</i>	<i>Facebook, Twitter, LinkedIn, Pinterest</i>
Communication	<i>Slack, Skype, Signal, Whatsdesk (WhatsApp), Discord, Viber</i>	<i>WhatsApp, Facebook Messenger, Google Chrome</i>
Productivity	<i>Calc (Excel), Impress (Power Point), Writer (Word), Draw (Visio), Gimp, PDF, Overleaf, Gmail, Thunderbird, Calendar, Dropbox, Box, Peek (Screen Recorder), Everpad (Evernote), Android Studio, GitKraken, Eclipse, VirtualBox, Toggl, Qualtrics</i>	<i>Gmail, Dropbox, Calendar, Todoist</i>
Travel & Local	<i>Airbnb, Google Maps, TripAdvisor, Expedia Travel, Uber, Yelp, Lyft, Grubhub</i>	<i>Google Maps, TripAdvisor, Uber, Yelp</i>
Health & Fitness	<i>WebMD, LiveStrong, MyFitnessPal</i>	<i>Google Fit, MyFitnessPal</i>
Entertainment	<i>YouTube, Angry Birds, Candy Crush, Spotify, Steam</i>	<i>Bitmoji, YouTube, Wordscapes</i>

Table 1: Summary of desktop (Ubuntu) and mobile (Android) applications tested for each category.

ory configurations and app mixes. A study from [29] suggests that the number of daily apps used by consumers is 10 with such consumers spending over 50% of their digital time on

Hardware Configuration	
Cores	4
ISA	x86-64, ARMv8 (64-bit)
Frequency	3.7GHz (x86-64), 2.1GHz (ARMv8)
IL1/DL1 Size	32KB
IL1/DL1 Block Size	64B
IL1/DL1 Associativity	8-way
IL1/DL1 Latency	2 cycles
Coherence Protocol	MESI
L2 Size	2MB
L2 Block Size	64B
L2 Associativity	16-way
L2 Latency	20 cycles
Memory Type	DDR4-2400 SDRAM
Memory Size	1GB, 2GB, 3GB, 4GB, 8GB, 16GB, 32GB

Table 2: Summary of hardware configurations.

Mix	Category Sequence
1	<i>social, communication, productivity, travel & local, health & fitness, entertainment</i>
2	<i>communication, productivity, travel & local, health & fitness, entertainment, social</i>
3	<i>productivity, travel & local, health & fitness, entertainment, social, communication</i>
4	<i>travel & local, health & fitness, entertainment, social, communication, productivity</i>
5	<i>health & fitness, entertainment, social, communication, productivity, travel & local,</i>
6	<i>entertainment, social, communication, productivity, travel & local, health & fitness</i>

Table 3: Summary of application mixes used for credential persistence testing.

mobile devices [30]. As such, our study assumes a baseline of 10 daily apps per platform for the average user.

On our desktop platform, we observed that the number of apps that could be launched while still having the ability to retrieve credentials ranged between 4 to 49 apps. On average, our firmware could locate login credentials after utilizing 22.3 apps. Figure 6(a) summarizes the persistence of credential information as a function of launched apps averaged across the six mixes from Table 3. Overall, the vulnerability factor (finding credentials) increased by 3x when doubling the memory size from 4GB to 8GB. However, this vulnerability factor plateaued once the memory capacity reached 8GB. For instance, systems with 8GB, 16GB, and 32GB memory capacities had the same average, tolerating 26.7 apps before the login credentials could no longer be found. A closer look at these results revealed that the gaming app, *Steam*, consistently caused our Facebook credentials to be evicted. This behavior was consistent across all the app mixes listed in Table 3. As a result, we conducted an additional experiment using our stress app to better characterize the vulnerability factor as a function of memory capacity. The results of this experiment revealed a strong correlation between memory capacity and the number of apps that could be used before credential eviction. We successfully located our password after launching 120, 279, and 570 apps on platforms configured with 8GB, 16GB, and 32GB, respectively. Even though applications, such as *Steam* can result in passwords being evicted, our overall attack is robust in desktop environments and can reliably retrieve credentials after running a variety of application mixes.

Figure 6(b) illustrates the results of the persistence experiment on our mobile platform. Overall, our attack performs the best against larger systems that have 3GB and 4GB memory capacities. We observed that the number of apps that could be used while still being able to retrieve Instagram’s credentials ranged between 1 to 12 apps depending on the memory size. On average, our firmware could locate login credentials after utilizing 5.7 apps. We observed a strong correlation between the device’s memory capacity and the number of apps used. This suggests that reasonably modern smartphones from 2015 that have larger capacities, such as Samsung Galaxy S6 and

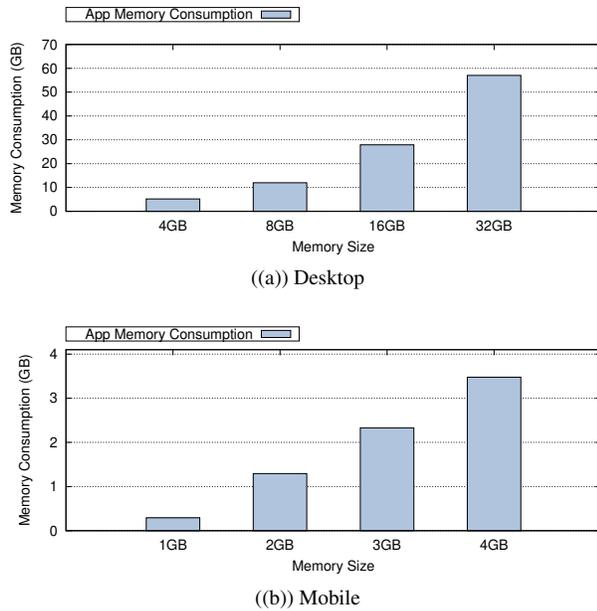


Figure 7: Summary of credential information persistence as a function of memory consumption.

HTC M9 are more prone to this attack. For instance, we observed that for a 1GB configuration, we could only retrieve the login credentials after 1 app has been used, with the password consistently deleted after a second app is launched. Further analysis shows that the vulnerability factor for a 2GB configuration increases to 3.2x relative to a 1GB configuration. The 2GB system allows for user credentials to still be detected after 2 to 5 apps have been used. The attack performs the best on the 4GB system showing a 10x increase in the relative vulnerability factor. The 4GB system allows for the login information to remain in memory after 7 to 12 apps have been used which is in line with the average number of daily apps consumers use. This is slightly higher than the 3GB configuration which remains vulnerable after using 8.8 apps.

5.3 Persistence vs. Memory Consumption

In order to evaluate the persistence of authentication information under different memory consumption profiles, we tested our attack with a configurable stress app. Using a stress app from [27] allowed us to systematically vary the amount of memory consumed in the system. Each experiment entailed first logging into Facebook/Instagram (Desktop/Mobile) and then launching the stress app. The memory footprint of the stress app was gradually increased in 100MB increments every minute. We then tested for the presence of the user’s credentials through firmware after every memory increment. We also ran multiple apps in the background including WhatsApp and Gmail to further stress the memory subsystem.

Figure 7(a) shows the results of the memory consumption

experiment for different memory capacities on our desktop platform. Although the number of launched apps has an impact on the persistence of credential information in memory, the persistence of such information strongly depends on the amount of memory a given app consumes. On average, we observed that for the 4GB system, the credentials remained in memory until the stress app consumed 1.2GB beyond the physical memory capacity. Evictions occurred after an additional 50%, 75%, and 78% of memory was consumed beyond the total physical memory of the system when configured with 8GB, 16GB, and 32GB, respectively. In other words, a significant amount of swap space had to be consumed before the credentials were evicted. Figure 8 illustrates this trend by showing the probability of finding credentials as a function of memory activity during our stress experiment on a 4GB system. The credentials on our desktop system remained in memory for 104 mins. Firmware failed to find the credentials after this 104 min duration which occurred after 900K pages have been evicted, as shown in Figure 8(a). We also observed that the system goes through different page-out rates. This is illustrated in Figure 8(c). We can see that at the beginning a steady page-out rate of about 3 pages per second occurred. However, a more aggressive page-out rate was observed before the credentials could no longer be found in memory.

Figure 7(b) shows the results of the memory consumption experiment for our mobile platform. Although the number of launched apps has an impact on the persistence of credential information in memory, the persistence of such information strongly depends on the amount of memory a given app consumes. On average, we observed that for the 1GB system, the credentials remained in memory when the stress app consumed less than 300MB. On average, this threshold increased to 1.3GB for a 2GB capacity, 2.3GB for a 3GB capacity, and 3.5GB for a 4GB capacity before the credentials were evicted. In general, Android’s MMU played a significant role in the retention of credentials with the MMU’s allocation policy varying as a function of memory capacity. The MMU allocated memory more aggressively for background apps as the capacity was increased from 1GB to 4GB. Furthermore, Android doesn’t use swap space. As such, background apps are terminated more frequently and the associated pages are freed. Unlike the desktop case, we can see in Figure 7(b) that password evictions occurred before the amount of physical memory was exhausted. Moreover, Figure 8 illustrates the probability of finding Instagram’s credentials as a function of memory activity during our stress experiment on a 4GB system. We observed that the credentials remain in memory for 74 mins. Firmware failed to find the credentials after this 74 min duration which occurred after 68K pages were evicted from memory. We found that Android frees 70K pages within 83 minutes compared to 310K pages in a desktop environment. Although the page-out rate in a desktop environment is 4.4x higher than what we observed for Android’s page-out rate, password evictions occurred much sooner on an Android

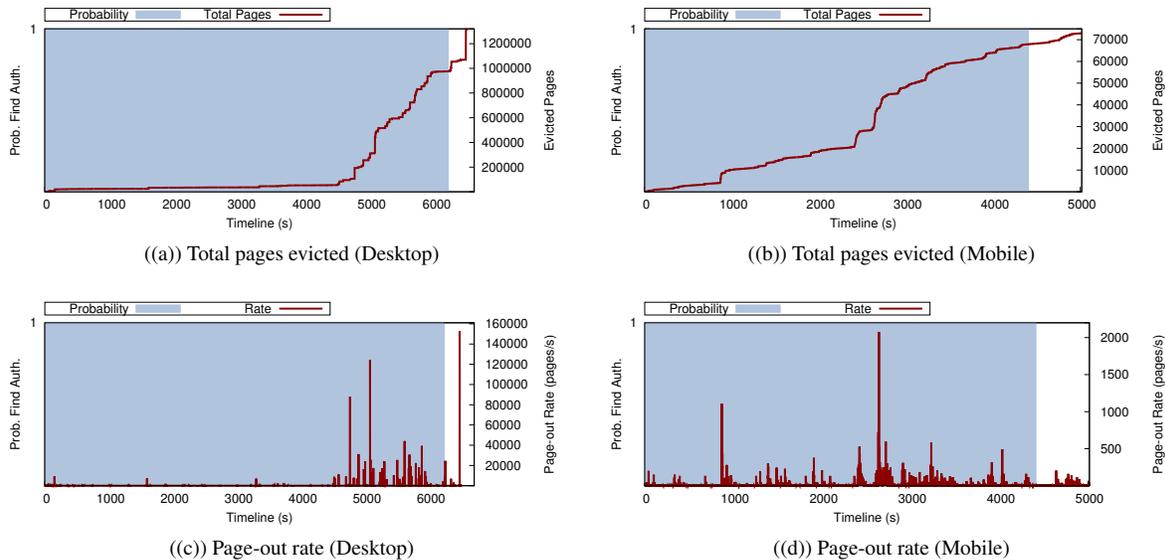


Figure 8: Probability of finding authentication credentials over time as a function of the total number of pages evicted (a) desktop (b) mobile and the page-out rate in pages per second for (c) desktop and (d) mobile.

platform. Figure 8(d) summarizes Android’s page-out rate over time. Unlike desktop systems, faster password evictions occurred in Android. We observed that Android initially freed pages at a relatively steady rate that is 3x higher compared to what we observed for a desktop system when excluding abruptly high page-out rates that are in excess of 1000 pages/s.

5.4 Performance Overhead

Our firmware-based attack is designed to selectively examine a limited number of pages in order to covertly extract sensitive data from HTTP requests without disrupting the normal execution of the system. Therefore, to better understand the suitability of our solution for desktops and mobile devices, we conducted runtime experiments across different applications and platform configurations. In this section, we examine the overhead of our solution compared to a solution that parses HTTP requests through full memory scans.

Figure 9 shows the number of searched pages as a function of launched applications and their respective runtimes. Since our previous experiments show that login credentials remain in memory well beyond 1 hour, we set up our management interrupts to ensure the completion of a full memory search cycle over a one hour period. Figures 9(a) and 9(b) summarize the number of user pages as a function of launched applications across our desktop and mobile platforms, respectively. We observe that the total number of pages that have their dirty, user/supervisor, and write flags set (D/U/W), varies as a function of launched apps and platform type. In general, our mobile platform running Android had 1.9x more D/U/W pages compared to an Ubuntu-based desktop platform. On

average, our firmware scanned 380 and 725 D/U/W pages for every desktop and mobile application, respectively. This correlates to each core spending a total of 180 μ s – 3.1 ms every hour (3 μ s – 52 μ s per minute) scanning memory when running 10 to 30 desktop apps and between 3.3 ms – 5.1 ms every hour (55 μ s – 85 μ s per minute) when running 10 to 30 mobile apps. The Ubuntu-based desktop overhead is relatively lower than the Android-based mobile overhead due to the reduced number of PTEs that have their D/U/W flags set. We also observe that by limiting our search to only D/U/W pages instead of all user pages, we eliminate scanning an additional 48K pages which corresponds to a 4.7x performance improvement, on average. Finally, we didn’t observe a significant difference between user pages that are writable and those that have become dirty. On average, we recorded across our tests 128 pages that were writable, but not dirty. While this figure is dependent on the application type, scanning pages that have their D/U/W flags as opposed to U/W presents another level of optimization for reducing the overhead of our attack.

Figure 10 illustrates the performance of an attack that uses a full memory scan approach and how it compares to our optimized solution. We measure the performance across different memory capacities, launched applications, and platforms. Figure 10(a) shows that the overhead increased linearly as a function of memory capacity when performing a full search. On our desktop system each core spent 23 s and 180 s scanning memory in order to complete a full search on 4GB and 32GB memory configurations, respectively. As such, this naïve approach doesn’t scale well to larger memory capacities. On the other hand, we observe that our page table-based attack sig-

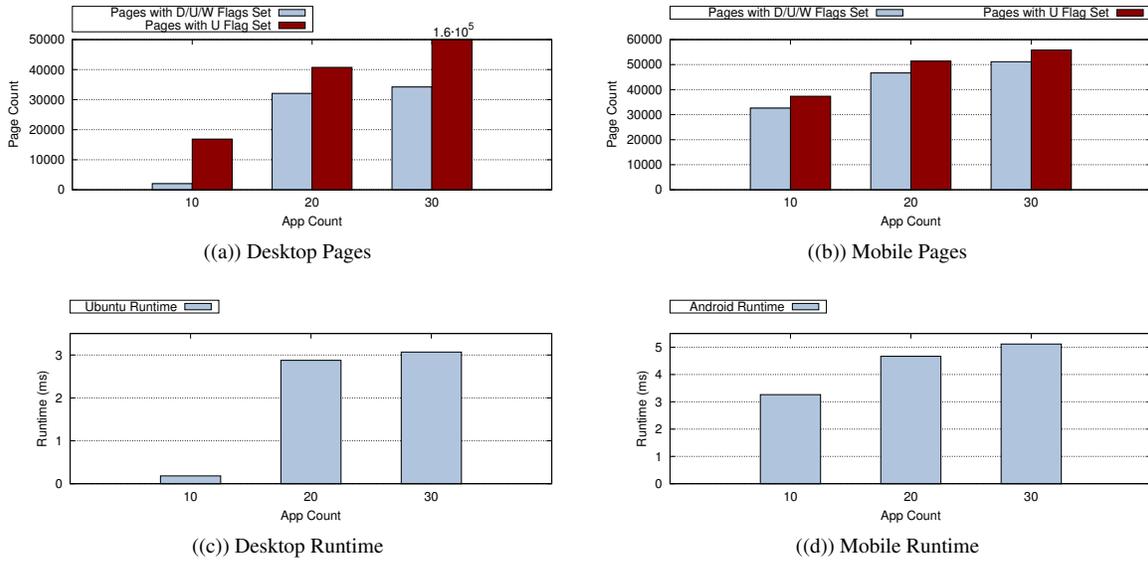


Figure 9: The number of searched pages as a function of launched applications (a) Desktop (Ubuntu), (b) Mobile (Android), and the runtime as a function of launched applications (c) Desktop (Ubuntu) and (d) Mobile (Android).

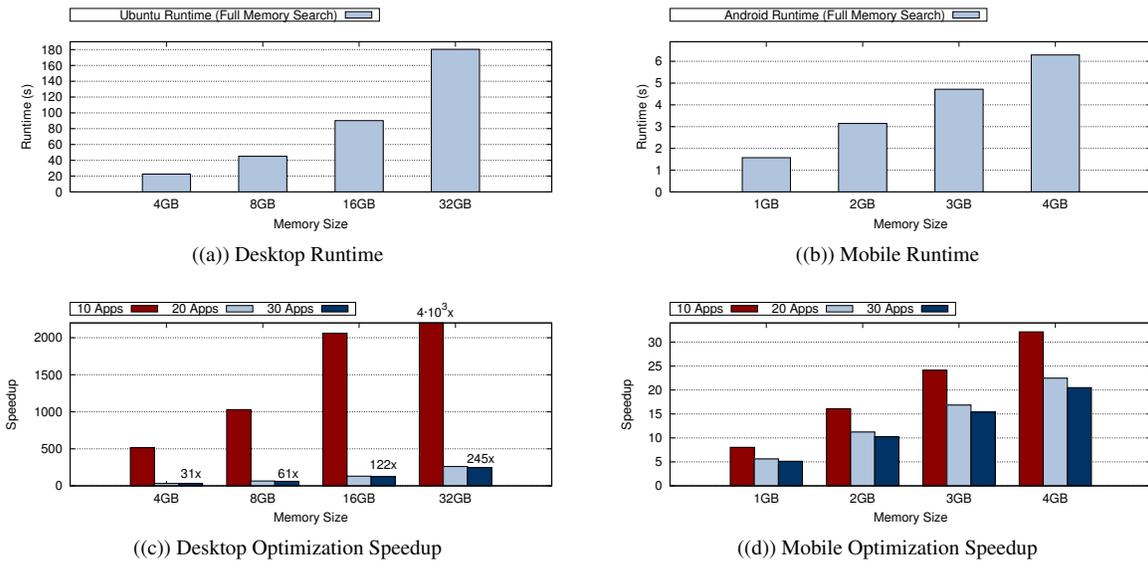


Figure 10: Runtime of full memory search as a function of memory capacity (a) Desktop (Ubuntu), (b) Mobile (Android), and speedup of our attack relative to a full memory search as a function of launched apps (c) Desktop (Ubuntu), (d) Mobile (Android).

nificantly improves the search time relative to a full memory search approach. We observe a relative speedup that ranges between 514x and $4.1 \cdot 10^3 \times$ when running 10 apps across memory capacities 4GB – 32GB on a desktop. The speedup was reduced to 31x and 245x when running 30 apps across memory capacities 4GB – 32GB on the same platform. This reduction was due to an increase in the number of candidate

pages our firmware scanned with 30 apps running on a desktop. This is summarized in Figure 10(c). Our results suggest a similar trend for mobile systems. This is shown in Figures 10(b) and 10(d). On our mobile system, each core spent 1.6 s and 6.3 s in order to complete a full search on 1GB and 4GB memory configurations, respectively. On the other hand, the speedup while using our optimization ranged between 8x –

32x when running 10 apps across memory capacities 1GB – 4GB and a speedup of 5x – 20x when running 30 apps across memory capacities 1GB – 4GB. Overall, our results show that our attack is suitable across mobile and desktop systems and shows that the overhead only increases as more applications are launched, irrespective of the memory capacity. The overheads of our optimized solution are shown in Figures 9(c) and 9(d).

5.5 Algorithm Robustness

We conducted several hours of testing on our firmware. The attack ran reliably and consistently under multiple workloads, load levels, and platforms. One of the stress experiments we conducted entailed evaluating the reliability of our firmware algorithm while searching memory in management mode. The experiment included continuously invoking firmware to search memory every one minute over a period of one week while we monitored the system for any crashes. The aforementioned steps were carried out on 8GB and 4GB systems that were already booted to the Ubuntu and Android OS's, respectively. The experiment also included interacting with each booted OS every one minute over an SSH connection. Both systems ran successfully without any crashes until the experiment was stopped.

6 Discussion

In this section, we discuss a possible countermeasure against firmware attacks. Since firmware operates at the lowest layer of the software stack, we consider a hybrid approach that entails the OS and hardware layers cooperatively restricting the processor from accessing non-firmware owned memory. To achieve this, the OS must register the memory ranges it will manage during its boot process, leaving out any regions already claimed by firmware. To support this, the hardware could expose a write-once table that we will refer to as the memory protection table (MPT).

During boot, the OS writes the memory ranges it owns into this table and validates that the intended ranges were properly programmed into the MPT. The OS raises a warning to the user in the event that it is unable to program the table as a possible indication that firmware has already programmed the ranges to purposely overlap with the OS's memory. Furthermore, any memory accesses issued to the ranges present in the MPT while the processor is executing firmware will result in an exception.

A challenge with the aforementioned approach is that the OS often relies on firmware to perform legitimate platform tasks on its behalf through runtime services. Since runtime services are in the form of code that belongs to firmware, it is conceivable that such services could be leveraged to collect secret information from memory. As a result, the OS must program the MPT with ranges that are owned exclusively by

the OS, excluding any ranges that are shared between the firmware and OS layers. This is because runtime services often require data to be exchanged with the OS through shared regions. Therefore, to support the countermeasure without breaking any legitimate runtime service functionality executed by firmware, we propose augmenting the processor's load/store queue (LSQ), the entity responsible for issuing memory transactions. In this case, the LSQ would serve the purpose of validating the program counter (PC) corresponding to the load/store instruction being executed and the address of the memory being accessed. To avoid triggering exceptions due to speculative instructions that may be squashed and never visible outside the processor, we propose waiting until a given entry within the LSQ is selected for retirement. At this point, the countermeasure checks the PC and the address of the memory transaction against the MPT containing the ranges originally programmed by the OS. An exception is raised if the PC corresponding to the load/store instruction isn't found in the MPT (not within the OS's exclusively owned range) and the address for the memory transaction is within the MPT (within the OS's exclusively owned range).

Detecting firmware accesses to memory through load/stores while the processor is in management mode would follow a similar approach to that of dealing with runtime services. When a management interrupt is issued to the processor, the mode will be set to reflect that it is in management mode. However, the same solution applies. The code executed in management mode belongs to firmware and as such would fall outside of the range programmed into the MPT. To this end, an exception would be raised in the event that the PC corresponding to the load/store instruction isn't found in the MPT and the address for the memory access is found within the MPT.

7 Related Work

Our related work is divided into: (1) prior work that relates to our attack, namely memory attacks and (2) prior work against the firmware subsystem that our proposed attack builds upon. **Memory Attacks.** For many years, system memory has been a prime target for stealing secret information [31–35]. In response to these challenges, researchers have explored various attacks that aim to expose the security risks associated with this subsystem [4–13]. Prior work [4] demonstrated the use of introspection through hypervisors for extracting credit card data from Point of Sale (PoS) systems that run as guest machines. However, such attacks are limited to environments that rely on hypervisors. Firmware attacks, on the other hand, are more dangerous, since firmware exists on every computing device. In addition, unlike hypervisors, firmware doesn't have the ability to trap memory accesses. As a result, our study explores a different set of challenges associated with covertly recovering data belonging to the application layer.

Other work examined the risks of memory scraping through

physical access [6–13]. For example, Halderman et al. [6] proposed using cold boot attacks to recover sensitive data from memory. The authors showed that a cooling agent could be applied to preserve the content of memory modules extracted from a stolen laptop, then later scanned on another machine to recover cryptographic keys. Muller et al. [36] proposed a similar approach against Android smartphones that could recover photos, personal messages, and passwords, in addition to cryptographic keys. Re-purposing available hardware ports is another technique that was used to collect memory dumps from smartphones. For instance, Munro [8] showed that the JTAG port on smartphones could be used to communicate directly with the system-on-chip and read out memory without the use of a cooling agent. Other attacks [9–13], considered installing malicious peripherals for reading arbitrary memory regions through DMA attacks. Virtually, all of these attacks require physical access or the presence of special hardware on the system. Unlike the aforementioned work, our attack is firmware-based and can be carried out remotely through software supply chain attacks or compromised live update utilities [1–3].

Firmware Attacks. Different forms of attacks against firmware have been explored [18–24, 37–53]. This includes crafted remote updates issued from user space to bypass safety measures such as secure boot [18–24]. For example, Kallenberg and Wojczuk [19] demonstrated the ability to overcome flash write-protection mechanisms and overwrite firmware at runtime by exploiting a race condition within the manageability chip. Other work [24], explored leveraging the presence of unused reference code that could be activated through integer overflows to initiate malicious firmware updates. Additional techniques include bypassing secure boot features by exploiting unsigned fragments such as company logo placeholders that are designated for OEM customization. Matrosov and Rodionov [54] demonstrated that malicious firmware could be launched in the presence of security features such as secure boot. Our work leverages the aforementioned attacks to insert untrusted firmware into the system and explore the risks associated with such malware. More importantly, we examine how such a component could be re-purposed to undermine the confidentiality guarantees the system provides to the application layer in the presence of such untrusted firmware.

8 Conclusion

In this work, we propose a novel firmware attack that leverages platform management cycles to extract sensitive data from HTTP requests. We develop a proof-of-concept of our attack using firmware that was configured to run with both Ubuntu 18.04 LTS and the Android Oreo 8.1. We show that a page table-based approach is sufficient to efficiently extract sensitive data from outgoing HTTP requests without disrupting the normal execution of launched apps. Our approach is up to $4 \cdot 10^3$ x faster compared to a full memory search imple-

mentation. Our attack claims limited execution cycles from the application layer making the attack difficult to detect even in mobile environments where app responsiveness is closely by consumers. Finally, we discuss countermeasures that could be employed to defend against such attacks.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive feedback and comments on this work. We extend special thanks to Yaohui Chen for shepherding our paper and working with us on producing the camera ready version of this paper. Finally, the authors would like to thank members of the Security and Systems Lab for all their support and feedback related to this work.

References

- [1] A. Greenberg, “Software has a serious supply-chain security problem,” 2017, <https://www.wired.com/story/ccleaner-malware-supply-chain-software-security>.
- [2] P. Gralla, “Malware takes aim at the supply chain,” 2018, <https://symantec-blogs.broadcom.com/blogs/expert-perspectives/malware-takes-aim-supply-chain>.
- [3] Kaspersky, “Operation shadowhammer,” 2019, <https://securelist.com/operation-shadowhammer>.
- [4] J. Hizver and T.-c. Chiueh, “An introspection-based memory scraper attack against virtualized point of sale systems,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2011, pp. 55–69.
- [5] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, “Ramble: Reading bits in memory without accessing them,” in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [6] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest We Remember: Cold Boot Attacks on Encryption Keys,” in *USENIX Security Symposium*, July 2008, pp. 45–60.
- [7] C. Hilgers, H. Macht, T. Müller, and M. Spreitzenbarth, “Post-mortem memory analysis of cold-booted android devices,” in *2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. IEEE, 2014, pp. 62–75.
- [8] K. Munro, “Android scraping: accessing personal data on mobile devices,” *Network Security*, vol. 2014, no. 11, pp. 5–9, 2014.

- [9] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, “Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals.” in *NDSS*, 2019.
- [10] U. Frisk, “Dma attacking over usb-c and thunderbolt 3,” Blog, October 2016, <http://blog.frizk.net/2016/10/dma-attacking-over-usb-c-and.html>.
- [11] —, “Direct memory attack the kernel,” *Proceedings of DEFCON*, vol. 24, 2016.
- [12] J. FitzPatrick and M. Crabill, “Stupid pcie tricks,” 2014.
- [13] U. Frisk, “Pcileech: Direct memory access attack software,” <https://github.com/ufrisk/pcileech>.
- [14] A. Bacha and R. Teodorescu, “Dynamic reduction of voltage margins by leveraging on-chip ECC in Itanium II processors,” in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 297–307.
- [15] —, “Using ECC feedback to guide voltage speculation in low-voltage processors,” in *International Symposium on Microarchitecture (MICRO)*, December 2014, pp. 297–307.
- [16] —, “Authenticache: Harnessing cache ECC for system authentication,” in *International Symposium on Microarchitecture (MICRO)*, December 2015, pp. 1–12.
- [17] S. Wall, “Resiliency for a cyber-physical future,” 2017, <https://www.slideshare.net>.
- [18] T. Hudson, X. Kovah, and C. Kallenberg, “Thunderstrike 2: Sith strike,” in *Black Hat USA*, 2015.
- [19] C. Kallenberg and R. Wojtczuk, “Speed racer: Exploiting an intel flash protection race condition,” 2015, white Paper.
- [20] R. Wojtczuk and C. Kallenberg, “Attacking UEFI boot script,” in *Chaos Communication Congress*, 2014.
- [21] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell, “Extreme privilege escalation on windows 8/uefi systems,” in *Black Hat USA*, 2014.
- [22] Y. Bulygin, A. Furtak, and O. Bazhaniuk, “A tale of one software bypass of windows 8 secure boot,” in *Black Hat USA*, 2013.
- [23] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell, “Senter sandman: Using intel TXT to attack bioses,” in *Hack in the Box*, 2014.
- [24] —, “Bios necromancy: Utilizing “dead code” for bios attacks,” in *Hack in the Box*, 2014.
- [25] N. Institute of Standards and Technology, “National vulnerability database,” 2019, <https://nvd.nist.gov>.
- [26] T. Spring, “First-ever uefi rootkit tied to sednit apt,” 2018, <https://threatpost.com/uefi-rootkit-sednit/140420/>.
- [27] A. Waterland, “Stress,” 2014, <https://people.seas.harvard.edu/apw/stress/>.
- [28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 Simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.
- [29] A. Annie, “Spotlight on consumer app usage,” 2017, <https://www.appannie.com>.
- [30] BusinessofApps, “App download and usage statistics,” 2019, <https://www.businessofapps.com/data/app-statistics>.
- [31] R. Inocencio, “New blackpos malware emerges in the wild, targets retail accounts,” Online, August 2014, <https://blog.trendmicro.com/trendlabs-security-intelligence/new-blackpos-malware-emerges-in-the-wild-targets-retail-accounts>.
- [32] L. Constantin, “Dexter malware infects point-of-sale systems worldwide, researchers say,” Online, December 2012, <https://www.csoonline.com/article/2132674/dexter-malware-infects-point-of-sale-systems-worldwide-researchers-say.html>.
- [33] D. Goodin, “Meet “chewbacca,” the point-of-sale malware that infected dozens of retailers,” Online, January 2014, <https://arstechnica.com/information-technology/2014/01/meet-chewbacca-the-point-of-sale-malware-that-infected-dozens-of-retailers/>.
- [34] S. Gatlan, “Dmsniff point-of-sale malware silently attacked smbs for years,” Online, February 2019, <https://www.bleepingcomputer.com/news/security/dmsniff-point-of-sale-malware-silently-attacked-smbs-for-years>.
- [35] N. Huq, “Pos ram scraper malware: Past, present, and future,” *A Trend Micro Research Paper*, pp. 1–16, 2014.
- [36] T. Muller and M. Spreitzenbarth, “FROST: Forensic Recovery of Scrambled Telephones,” in *International Conference on Applied Cryptography and Network Security*, June 2013, pp. 373–388.

- [37] F. Cao, Q. Li, and Z. Chen, “Vulnerability model and evaluation of the uefi platform firmware based on improved attack graphs,” in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2018, pp. 225–231.
- [38] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, “Trustice: Hardware-assisted isolated computing environments on mobile devices,” in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.
- [39] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for linux-based embedded firmware.” in *NDSS*, 2016, pp. 1–16.
- [40] J. Taylor, B. Turnbull, and G. Creech, “Volatile memory forensics acquisition efficacy: A comparative study towards analysing firmware-based rootkits,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, 2018, p. 48.
- [41] F. Zhang, H. Wang, K. Leach, and A. Stavrou, “A framework to secure peripherals at runtime,” in *European Symposium on Research in Computer Security*. Springer, 2014, pp. 219–238.
- [42] M. LeMay and C. A. Gunter, “Cumulative attestation kernels for embedded systems,” *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, 2012.
- [43] D. Schellekens, P. Tuyls, and B. Preneel, “Embedded trusted computing with authenticated non-volatile memory,” in *International Conference on Trusted Computing*. Springer, 2008, pp. 60–74.
- [44] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, “Mouse trap: Exploiting firmware updates in {USB} peripherals,” in *8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14)*, 2014.
- [45] Y. Li, J. M. McCune, and A. Perrig, “Viper: verifying the integrity of peripherals’ firmware,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.
- [46] D. Peck and D. Peterson, “Leveraging ethernet card vulnerabilities in field devices,” in *SCADA security scientific symposium*, 2009, pp. 1–19.
- [47] A. M. Garcia Jr, “Firmware modification analysis in programmable logic controllers,” AIR FORCE INSTITUTE OF TECHNOLOGY WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF . . . , Tech. Rep., 2014.
- [48] K. Chen, “Reversing and exploiting an apple firmware update,” *Black Hat*, vol. 69, 2009.
- [49] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, J. Blocki, K. Fu, and D. Song, “Take two software updates and see me in the morning: The case for software security evaluations of medical devices.” in *HealthSec*, 2011.
- [50] C. Miller, “Battery firmware hacking,” *Black Hat USA*, pp. 3–4, 2011.
- [51] B. Jack, “Jackpotting automated teller machines redux,” *Black Hat USA*, 2010.
- [52] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, “Comprehensive experimental analyses of automotive attack surfaces.” in *USENIX Security Symposium*, vol. 4. San Francisco, 2011, pp. 447–462.
- [53] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, “Experimental security analysis of a modern automobile,” in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.
- [54] A. Matrosov and E. Rodionov, “Uefi firmware rootkits: Myths and reality,” in *Black Hat Asia*, 2017.

A Framework for Software Diversification with ISA Heterogeneity

Xiaoguang Wang
Virginia Tech

SengMing Yeoh
Virginia Tech

Robert Lyerly
Virginia Tech

Pierre Olivier
The University of Manchester

Sang-Hoon Kim
Ajou University

Binoy Ravindran
Virginia Tech

Abstract

Software diversification is one of the most effective ways to defeat memory corruption based attacks. Traditional software diversification such as code randomization techniques diversifies program memory layout and makes it difficult for attackers to pinpoint the precise location of a target vulnerability. Some recent work in the architecture community use diverse ISA configurations to defeat code injection or code reuse attacks, showing that dynamically switching the ISA on which a program executes is a promising direction for future security systems. However, most of these work either remain in a simulation stage or require extra efforts to write program.

In this paper, we propose HeterSec, a framework to secure applications *utilizing a heterogeneous ISA setup composed of real world machines*. HeterSec runs on top of commodity x86_64 and ARM64 machines and gives the process the illusion that it runs on a multi-ISA chip multiprocessor (CMP) machine. With HeterSec, a process can dynamically select its underlying ISA environment. Therefore, a protected process would be capable of hiding the instruction set on which it executed or detecting abnormal program behavior by comparing execution results step-by-step from multiple ISA-diversified instances. To demonstrate the effectiveness of such a software framework, we implemented HeterSec on Linux and showcased its deployability by running it on a pair of x86_64 and ARM64 servers, connected over InfiniBand. We then conducted two case studies with HeterSec. In the first case, we implemented a multi-ISA moving target defense (MTD) system, which introduces uncertainty at the instruction set level. In the second case, we implemented a multi-ISA-based multi-version execution (MVX) system. The evaluation results show that HeterSec brings security benefits through ISA diversification with a reasonable performance overhead.

1 Introduction

Software diversification has proven to be a very effective way to defeat software memory corruption attacks [42]. By

diversifying the target application memory layout, these diversification techniques are capable of randomizing vulnerable code locations [3, 8, 35, 36, 40, 63, 78, 81], detecting abnormal program behaviors (i.e. attacks) [15, 38, 51, 57, 58, 72, 73, 83], or hiding the secret data [39, 41]. The uncertainty brought about by a diversified program effectively raises the bar for launching a successful attack.

The “end of Moore’s Law” [21, 25] has forced chip vendors to advance performance and energy efficiency boundaries elsewhere, in particular by designing radically different hardware: multicore and manycore chips [11, 56, 61]; CPUs with heterogeneous micro-architectural properties [34, 53], partially overlapping ISAs [32], and various forms of accelerators and programmable hardware [22] that exploit heterogeneity. CPUs with heterogeneous-ISA cores – studied by the academic research community [4, 44, 52, 69, 71] – are another point in the architectural design space that are now available as commodity hardware – e.g., Intel Skylake processor with in-package FPGA [28, 29] enables synthesizing RISC-V and x86 soft cores; AMD’s new generation x86 processor integrates ARM cores; commodity smart NICs integrate ARM [24, 49], MIPS64 [64], or Tile cores [48].

Recently, some research efforts explored using multiple, heterogeneous-ISA CPUs to secure the application execution. For example, architecture researchers have proposed systems that implement heterogeneous ISAs over one single chip to achieve inter-ISA program state randomization with higher entropy [70, 71, 76]. Another recent work leverages the distributed, heterogeneous-ISA machines to detect program vulnerability exploits [73]. Specifically, it simultaneously runs multiple instances of the same application on heterogeneous-ISA machines to detect execution divergence caused by eventual security attacks (a.k.a., multi-variant execution [15]). However, programming on such a distributed, multi-ISA environment is not easy, as it requires tremendous efforts to synchronize program states over differential OS kernels and instruction sets.

In this paper, we make the first step towards applying software-based diversification concepts to a real multi-ISA en-

vironment, aiming to secure software execution with ISA heterogeneity. An ISA-diversified program can have additional randomness in its code and data memory layout, register usage, instruction orders, and micro-architecture behaviors. Furthermore, the diversified program variants could potentially leverage some architecture-dependent security extensions, making it even harder for attackers to bypass a single layer of protection. To achieve this goal, we propose HeterSec, a software framework that facilitates securing applications with multiple ISAs. Unlike existing simulation-based approaches, HeterSec bridges real heterogeneous-ISA machines. HeterSec works at the operating system and process runtime level, giving processes an illusion of running on a CMP machine while possessing the ability to dynamically select the underlying instruction set or cross check program state between two ISA-diversified program instances.

To demonstrate its effectiveness, we have built two security applications on top of HeterSec. The first security application enables the target program to randomly execute between machines with different ISAs, implementing a moving target defense (MTD) system [31]. The second security application implements a multi-variant execution (MVX) system [15, 38]. A traditional MVX system runs multiple variants of an application with non-overlapping address space [15, 83]. On detecting abnormal runtime behaviors from the variants (e.g., unmatched system call return values, segfault), a MVX monitor could deduce there is likely an ongoing exploit. Variants generated from the ISA heterogeneity can automatically obtain an additional level of diversity, making attackers even harder to successfully launch an attack. Overall, we explored the research space in securing software execution with diversified instruction sets. To that aim, we made the following contributions:

- We built a software framework that can manage the process execution over coupled multi-ISA machine nodes for security purposes.
- We implemented two security applications on top of such a framework, namely multi-ISA based MTD and multi-ISA based MVX. The multi-ISA MTD randomly changes the execution ISA, hiding the precise target hardware features from attackers. The multi-ISA MVX uses ISA diversity as an additional dimension to differentiate program instances so that it is even harder for attackers to bypass the violation check.
- We demonstrated the potential of such multi-ISA based security systems with real-world evaluation; the results show that the additional layer of ISA diversity increases the cost for attackers, adding about 15% overhead for Nginx and Redis server applications in real-world scenarios.

The rest of this paper is organized as follows: Section 2 provides some background information of multi-ISA systems.

We then describe the design, implementation and case studies of HeterSec in Section 3. The evaluation is presented in Section 4. Afterwards, we summarize the related works in Section 5 and conclude the paper in Section 6.

2 Background and Threat Model

In this section, we briefly introduce the background on moving target defense and multi-variant execution; next we describe our motivation by summarizing recent multi-ISA systems and the security implications; we then proceed to define the threat model at the end of this section.

2.1 MTD and MVX

Moving Target Defense (MTD) Most information systems are built on relatively static platforms. Many defense techniques also involve static integrity checks and introspection. The static nature of such defense mechanisms gives attackers the time to thoroughly study the target system and launch the exploit [31]. The goal of MTD is to break the static nature of the target systems, with deviation of existing defense mechanisms and adaptations over time. MTD is an abstract concept, leaving options open with regards to how it is implemented. Thus realization of its design philosophy can be demonstrated in many ways. For example, dynamic systems or network configuration [30, 31], dynamic application code and data [13, 81], etc. In this paper, we demonstrate the security benefit and performance cost of running processes with dynamic execution on multiple ISAs using HeterSec.

Multi-Variant eXecution (MVX) Another interesting way to secure applications with multiple ISAs is through multi-variant execution systems [15]. MVX is a software security technique that runs multiple functionally equivalent programs (variants) with differing memory layouts. Some examples of such memory layout differences and deltas include non-overlapping memory maps [38, 57, 83], reverse stack growth [58], etc. By executing the diversified variants with the same inputs, the MVX engine is capable of detecting when an attack happens if one of the variants fails. That being said, existing MVX techniques might not be met with as much success when attempting to detect attacks based on relative addresses [23, 27, 73] or architecture level vulnerabilities [37, 46, 82]. With HeterSec, we built a prototype to use multiple ISAs as the source of variation between variants and prove a multi-ISA MVX system is still capable of obtaining reasonable performance despite the overheads involved.

2.2 Multi-ISA Systems and Security

Heterogeneous CPUs have been widely adopted in both data centers and end devices. On mobile platforms, ARM big.LITTLE technology uses two types of processor to

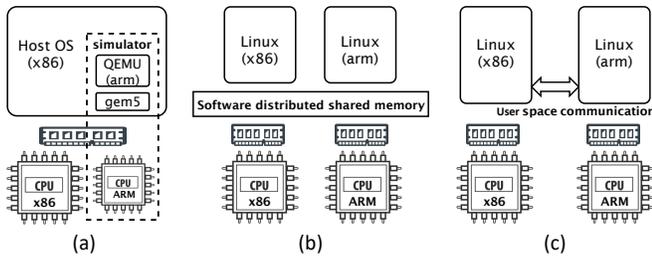


Figure 1: Comparison of multi-ISA security systems: (a) HIP-StR with simulated multi-ISA chip [70, 71]; (b) HeterSec; (c) DMON on completely decoupled machines [73].

achieve a dynamic balance between maximum power efficiency and maximum compute performance [1]. On data center servers, heterogeneous ISA processors are being used in different scenarios. For example, GPUs and TPUs are often equipped to accelerate machine learning workloads [33]. ARM based PCIe-pluggable SmartNIC cards are used to offload network applications for improved throughput and security [67, 68]. In academia, there have been several works exploring the benefit and cost of building single-chip multi-ISA systems [20, 70, 71]. DeVuyst et al. [20] first demonstrated the possibility of building a multi-ISA chip on a simulator, showing the ability to migrate processes between different ISAs. Venkat et al. [71] further expanded on the idea by proposing that applications running on multiple ISA could have benefits in reducing power consumption and accelerating computation speed, which they called *ISA affinity*. Their findings proved that an application can have lower power consumption (or better performance) by being split into code phases. Based on the ISA affinity of each code phase, the application code can be selectively executed across a heterogeneous ISA chip. In terms of security, HIPStR explores using multiple ISAs to increase code entropy, which makes return oriented programming (ROP) attacks difficult to launch [70]. The difference between these works and our system is that they are all built on top of CPU simulators (gem5 [9] and QEMU [55] as shown in Figure 1 (a)). The simulation-based approach makes it hard for security researchers to investigate the security benefits of using a multi-ISA architecture.

A recent concurrent work, DMON [73], uses distributed heterogeneous-ISA machines to generate and host program variants (a distributed version of N-Variant Execution [15]). The variants run on completely separate machine nodes and each variant communicates with the counterpart variant through a lightweight UDP-based network protocol (Figure 1 (c)). Although lightweight network protocols can provide low latency communication cost to exchange data between distributed nodes, the use of `ptrace` interface to intercept system calls brings extra context switches. For example, running Lighttpd web server on DMON will have 5.43x performance overhead [73]. Furthermore, DMON does not provide

generally sufficient abstraction to secure applications with multi-ISA machines. Therefore, it may provide less extensibility for developing multi-ISA based security applications which require timely execution of ISA switches. HeterSec instead focuses on building a generic framework to secure applications with the multi-ISA architecture. To this end, HeterSec adopts a hybrid approach – it runs on top of the real ISA-heterogeneous hardware; but the protected process has a unified view of system resources as if it runs on a multi-ISA CMP platform (Figure 1 (b)).

3 Design and Implementation

3.1 System overview

HeterSec aims to secure process execution by utilizing ISA heterogeneity to, for example, randomize the process execution environment over heterogeneous machines. To achieve this, HeterSec provides a per-process HeterSec execution environment. Specifically, it allows the protected process to be executed on machines running with different ISAs as if it were running on a single machine.

Figure 2 shows an overview of HeterSec with its new components added to an existing computer system stack. The components introduced by HeterSec include both the kernel and the user-space runtime as shown in blue. Figure 2 also illustrates two security application scenarios on top of HeterSec. In the first scenario, HeterSec switches the underlying ISA out from under the protected application, increasing the entropy of possible program states by masking the ISA switch and preventing attackers from divining underlying hardware details. In the second scenario, HeterSec launches multiple variants of the program, monitors the variants’ execution states (e.g., return values of system calls, or segfault), and raises an alert on any execution divergence caused by a potential attack. The HeterSec kernel provides additional functionality to control the target process at runtime, such as the process interception, per process shared memory and fast inter-kernel messaging. For example, the HeterSec distributed operating system kernel maintains a synchronized page table for each protected process. The page tables are synchronized during each ISA switch, giving the HeterSec process a unique view of the memory. Secure application scenarios can be implemented as loadable kernel modules that interact with the target process execution. Since HeterSec only intercepts and interacts with the target process, it introduces nearly zero performance overhead to other processes running on HeterSec ¹.

HeterSec has a concept of master OS. The master OS is the OS where the HeterSec process is initialized and launched. Correspondingly, the OS that works as the counterpart to the master OS is called the follower OS. The master HeterSec OS exports the view of system resources to the HeterSec process

¹Except for a few in-kernel checks to verify the process status, HeterSec kernel does not bring extra code paths for non-HeterSec processes.

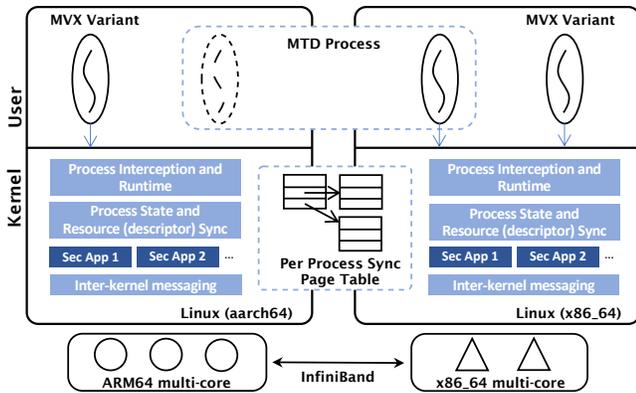


Figure 2: The architecture overview of HeterSec with two security application scenarios. The components in blue indicate modifications over existing software stack.

running on follower OS. Such system resources are often unique for each process, for example, open file descriptors, sockets, or event poll descriptors. For consistency reasons, HeterSec has to ensure that only one copy of such resources is maintained across OSes, maintaining a single source of truth for in-kernel state. When necessary, the master OS also helps to initialize and build the virtual address space for the follower OS’ process. Virtual memory areas (VMAs) and pages are synchronized between the two OSes. All the inter-OS communication requests are registered with an inter-kernel messaging API so that messages can be less expensive as they avoid going through the complicated network stack. All the software components mentioned above are running on multi-ISA machines, connected over InfiniBand. In the following section, we describe the details of each component.

3.2 HeterSec distributed kernel

The HeterSec distributed kernel can be considered as a special implementation of multikernel systems [7]. Instead of running on a multi-core NUMA machine, HeterSec runs on a *heterogeneous ISA multi-domain “machine”*, with each computing domain connected with a high-speed network connection. By using this approach we could avoid to use simulation or dynamic code translation, so that code can be executed at nearly native speed. However, there are two problems with such a heterogeneous ISA multi-domain “machine”: first, there is no memory coherence guaranteed between multi-ISA nodes, which raises programmability issues if we intend to leverage the heterogeneous ISA, multi-domain capabilities to implement security applications. Second, it is hard to manage the distributed resources (e.g., opened descriptors, network connections) on top of the heterogeneous instruction sets.

To solve those issues, HeterSec does not maintain global state for all OSes, but instead chooses only to maintain some HeterSec process specific states, synchronizing them on de-

mand. To be compatible with existing software stacks, the HeterSec distributed operating system is designed as several kernel extensions and is built based on the Linux kernel. There are three major components that facilitate HeterSec processes running on heterogeneous ISA machines: the per-process page table synchronization, secure applications, and the system resource sharing service.

The first component is a per-process page table synchronization handler. HeterSec provides a synchronized page table for each HeterSec process. The state is then synchronized across the x86_64 and ARM64 machines on demand. Before the process is started as a HeterSec protected process, the secure application (a kernel module) has to be loaded and subsequently pass the defined security policy to the process runtime. Such security policies include how frequently to switch the instruction sets, which system calls are used to synchronize and check the program states in the multi-variant execution. The runtime then executes the protected process accordingly - for example, randomly running processes across multi-ISA nodes or concurrently executing variants with cross-ISA lockstep state checking. In short, based on the secure application scenario, the HeterSec kernels maintain the synchronized memory views across the multi-ISA nodes. In the current design, HeterSec leverages a dedicated kernel thread to synchronize the pages in the background. It maintains a simple *read-duplicate write-invalidate* protocol for the shared memory pages [80].

Another essential component for HeterSec is the system resource sharing service. HeterSec maintains a single view of the system resource from the HeterSec process perspective. That means for each HeterSec process, there should be only one set of the network sockets, opened file descriptors, etc. Unfortunately, system resources such as file descriptors, sockets and event descriptors, are difficult to be shared across machine boundaries due to the difficulty in splitting the in-kernel state. One potential solution could be using a Network File System (NFS) to share and synchronize the file systems across the OSes. However, this will introduce potential issues for those pseudo-files located in `/dev/tty`, or `/proc`. Architecture dependent shared libraries also use different instruction formats and EFL binary contents. Naively synchronizing those files will cause runtime errors and crashes in these programs. To address this problem, HeterSec combines an implementation of system resource remote procedure call (RPC) and a virtual descriptor table (VDT).

Before starting the process, the secure target application can be specified with a white list of files that should be loaded locally. By default, we put the standard output (i.e., `stdout` and `stderr`), the shared libraries and configuration files in the white list. During the protected process’s runtime, the follower OS will build up a VDT. For each table entry, it indicates whether a descriptor should be accessed locally or remotely on the master node. For instance, we do not want to create two sockets for a single connection request on HeterSec

MVX. Therefore, the HeterSec kernel running on follower OS have to simulate the socket creation by placing a fake socket descriptor in the virtual descriptor table and mark that entry as virtual (V). On the contrary, descriptors of the opened library files and a `stdout` are marked as real (R) as they should be accessed locally. For system resource requests that have to be handled on the master OS, a system call RPC mechanism is provided. A system call server on the master OS handles the remote system call request, sets up the buffer value on the virtually shared pages, and returns the result to the caller on the follower node. The virtually shared pages are synchronized between nodes by the HeterSec kernel, as mentioned above.

We also support some termination signals (e.g., `SIGINT`) on the master node. On receiving a termination signal during the remote system call context, the master side HeterSec kernel replies a negative system call return value (i.e., `-ERESTARTSYS`) back to the follower kernel. The follower kernel then stops the HeterSec processes on the follower node. When the termination signal comes within the master kernel context, the master forwards the signal to the follower. Correspondingly, the follower terminates the execution loop. The master then stops itself by calling `do_exit()`.

3.3 Handling the cross-ISA code execution

Executing code on the multi-ISA “machine” as if on a single machine is challenging, since it requires several architecture-dependent code generation and state exchanges. HeterSec requires the architecture-dependent binaries generated from the same source code (e.g., same application source code and library code). This can unify most of the cross-ISA code execution behaviors, such as system call sequences. The generated binaries contains all the necessary information to run a protected process across ISA-different nodes. This information consists of instructions and data emitted by the compiler for each ISA. It may also carry some additional information such as the program state transformation routines. The types of information are decided by each individual security scenario. For example, cross-ISA randomized MTD execution would require information to transform the execution state from one architecture to another. This is because fine-grained program state (e.g., the variables on stack) must be synchronized accordingly as each architecture has its own specification for stack layout and register usage (Section 3.4.1). Security applications such as multi-ISA MVX require less information in metadata as each program instance is mostly self-contained on a single machine. The system call parameters (i.e., userspace buffers) and the opened descriptors are synchronized by the distributed operating system kernels mentioned above. It simplifies the system call simulation which is commonly used in existing MVX techniques (Section 3.4.2).

HeterSec introduces a new system call (i.e., `sys_hscall`) to identify the protected process and enable it to run on multi-

ISA nodes. That system call sets up a bit in the process descriptor (`task_struct` in Linux); after that, the HeterSec code path in the distributed kernels will be triggered to support cross node process execution. For example, when defending the protected process in the MVX mode, we can initiate that system call to launch two process variants on both nodes. The in-kernel MVX engine checks the system call sequences and return values and raises an alert on any execution divergence. More details are discussed in Section 3.4.2.

3.4 Case Studies

We have built two security application scenarios on top of HeterSec, which can fully utilize the instruction set diversity.

3.4.1 Multi-ISA MTD

The first security application is a heterogeneous-ISA based MTD system. Unlike most existing MTD or code randomization techniques [19, 30, 31], HeterSec randomizes the code execution path by switching ISAs at runtime. From the protected process’s perspective, it runs on top of a dynamic hardware environment with ISA diversity. Therefore, it would be hard for an attacker to prepare the exploit payload, for example, finding the correct ROP gadget chain or accurately measuring the hardware timing for side-channel attacks. When the process execution encounters a potential ISA switching point, the runtime will randomly decide which ISA the process will execute on in next step. Those ISA switching points are similar to the randomization points in existing code re-randomization works [8, 81], except existing randomization techniques update the code pointer references while HeterSec updates the architecture related states (e.g., stack slots, register set). Although the implementation sounds straightforward, there are some subtle issues when implementing such a system on multi-ISA architecture.

Pointer and architecture specific structure handling is one such case. Some system calls return with data updated to the userspace. Linux handles block data copying between userspace and kernel with pointers and helper functions such as `copy_to_user()`. When calling a system call across nodes, the follower OS context has to make sure any userspace memory updates are synchronized to the local node. In our implementation, most of the userspace memory updates caused by remote system calls will be synchronized correctly with the help of on-demand page synchronization. However, we noticed that Linux maintains slightly different format of some data structures on ARM64 and x86_64. The `struct stat` and `struct epoll_event` are two such cases. The `struct epoll_event` on x86_64 Linux is enforced to have the same alignment as that structure in 32-bit Linux (with packed attribute) in order to make 32-bit simulation easier. On the other hand, the ARM64 kernel does not enforce such alignment.

To solve this issue, we converted the structure formats in

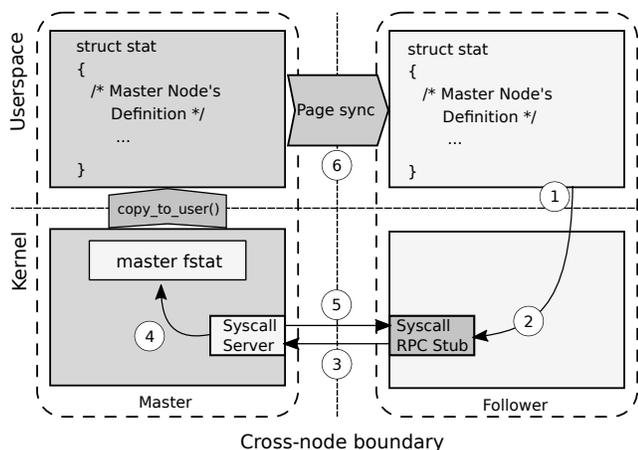


Figure 3: Program flow for an example `fstat` system call executed on the follower node.

`musl-libc` headers on the follower node to mimic the layout of the master’s format. When the follower OS issues a system call RPC, the master OS handles the request and updates the memory references in its own address. The page synchronization handler forwards the change to the follower OS. Figure 3 represents the different stages a system call like `fstat` goes through when called on the follower node. First, the syscall enters the kernel on the follower and calls the RPC stub in Step 2. This RPC stub then communicates to the syscall server on the master node over the messaging API in Step 3. Next, in Step 4 the syscall server calls `fstat` on behalf of the follower, which subsequently completes and returns to the stub as shown in Step 5. It also copies the data to the master’s userspace memory. This userspace data is synchronized to the userspace of the follower through page synchronization shown in Step 6 to maintain the illusion to the user program that its own kernel performed the syscall operation.

Randomization and transformation library: As a working proof of concept for the Multi-ISA MTD idea, we implemented an MTD randomization library. It makes decisions on whether a process should execute on a particular node based on random numbers generated from `/dev/urandom`. The probability threshold is also read in from a configuration file (in root mode) at runtime. This enables us to modify MTD switch probability without needing to recompile the application. We leverage the transformation library in Popcorn compiler framework to transform the code execution states between multi-ISA nodes [4].

ISA-switching point insertion, modular compilation, and MTD region activation: When compiling these production level applications the main goal was to make the process as unobtrusive as possible, only generating required metadata in relevant or vulnerable functions and files. The HeterSec framework allows the user to compile specific source files and generate stack transformation metadata for only those sources,

effectively enabling MTD functionality for certain parts of the program. While this functionality only works on a file-based granularity, one step of the compilation process includes an LLVM pass to add in calls to the randomization library mentioned in the previous section. These calls are added to the instrumented function prologues and epilogues and can be individually activated on each architecture through configuration files specifying the individual call-sites to activate. Once activated, whenever the program enters or exits these functions it checks with the MTD randomization library if it should switch ISAs, giving us granularity at a function level. We select the source code that contains the critical path for most of the work (e.g., the event loops), and compile them with ISA-switching points instrumented. For example, on Nginx we selected function call paths like `ngx_process_events_and_timers()` in `event/ngx_event.c`. For Redis, we selected similar functions in event loop path, for example `processTimeEvents()` located in `ae.c` that calls the `serverCron()`. These functions are called at a frequency of `server.hz` which defaults to 10 hz. By targeting where we place these checks across the program through this modular compilation, we avoid unnecessary calls to the MTD randomization library, reducing the overhead of the HeterSec framework.

3.4.2 Multi-ISA MVX

The second security application is a heterogeneous-ISA based multi-variant execution system. Similar to a traditional MVX system, the HeterSec MVX also has one leader variant and one follower variant. The leader runs on the master OS with full access to system resources, while the follower is only allowed to execute computational and memory-related code. Since there is only one valid copy of system resources (i.e., opened file, socket and event descriptors), the MVX engine should have the ability to guarantee two program variants can execute simultaneously over a single set of system resources. The HeterSec MVX engine uses system call simulation to synchronize the state across the variants. Specifically, an MVX monitor intercepts the selected system calls from the leader variant’s execution and forwards the system effects (e.g., memory update) to the follower. The MVX monitor verifies the system call return values between the running variants and also captures any memory fault to detect divergent (and potentially malicious) behaviors.

In HeterSec, the MVX engines are located inside each distributed kernel as shown in Figure 2. At runtime, the MVX engine on the follower OS verifies whether a system call should be simulated or directly passed through to the local kernel. For system calls tagged for pass-through, the follower OS serves the HeterSec process as usual. For system calls that access the per process descriptors, the HeterSec runtime will verify the descriptor against the virtual descriptor table (described in Section 3.2). Currently, sockets and event poll descriptors are marked as virtual descriptors, meaning that

those descriptors accesses will be simulated on the follower variant by replaying the system call effects from the master OS. For example, the MVX engine will simulate the system call `sys_recvfrom(int sockfd, void *ubuf, ...)` for the follower variant by coping the `ubuf` data from the leader variant to the `ubuf` address in the follower variant. Unless specified, file descriptors by default are marked as real descriptors and they are accessed locally. Similar to the files handled in MTD scenario, variants executed on heterogeneous-ISA nodes have to load shared libraries in different ELF formats. We require the user to manually copy all the necessary files before starting the application as a MVX process. This procedure can also be made automatic by using a NFS to synchronize these files across nodes.

There are other subtle issues when implementing MVX on multi-ISA nodes. One issue is the default libc libraries on two nodes could potentially cause differing system call sequences. To prevent false positives, we compile the application source code and link the object files with the musl libc library generated from the same source. As a result, the system call sequences are almost the same across the binaries on different architecture, with the exception of a few thread initialization functions such as `set_tid_address(int *tidptr)`. In this case, we just ignore performing comparisons on such system call executions. In addition, the system call numbers (and some names) are different in ARM64 and x86_64 architectures and therefore cannot be directly mapped across multi-ISA nodes. For instance, the ARM64 Linux kernel has replaced the `open` system calls with `openat`. This is also the case for several other system calls with "at" suffix. Consequently, we cannot forward a system call directly across machines using its number on any particular architecture. Instead, we maintain a system call number translation table, so that any system call (number) being sent to the counterpart node for simulation will be translated to the corresponding system call number first. In our current MVX engine implementation, we do not handle multi-threaded applications. However, we believe a deterministic multi-threading library [47] can be used in HeterSec to solve that issue.

We implemented two types of multi-ISA MVX where the monitor resides (1) in a separate process using `ptrace` to check the application and (2) in the kernel as a Linux kernel module. The `ptrace`-version MVX monitor is used as the developing mode, as it is easier to debug the monitor code without rebooting the operating systems. Similar to some existing MVX works, the `ptrace` version MVX uses the `ptrace` parent process as the MVX monitor, leveraging `ptrace` primitives to intercept and simulate the system calls. It implements a shared ring buffer to pass events (e.g., the syscall return values, or the modifications of data structures) between nodes using a FIFO queue policy in order to maintain sequential consistency. The kernel module version of MVX can be used for deployment because of the better performance. The MVX engine in the kernel intercepts the system calls by

wrapping and instrumenting the system call handlers. It also registers the MVX engine code with the HeterSec message layer for fast cross-node messaging. As a result of moving the implementation inside the kernel scope, the system call interception cost and communication latency are both lower than the `ptrace`-based prototype (in Section 4.2).

3.5 Implementation

We implemented a prototype of HeterSec on a x86_64 and ARM64 machine pair, connected using a Mellanox ConnectX-4 InfiniBand network. The synchronized address space was implemented by placing hooks in the page table handler in the kernel virtual memory subsystem (e.g., hooking the `vma` and `pte` operations [17]). When the protected process is executed on the follower OS, the follower OS kernel handles the page fault by fetching pages from the master OS. The master OS kernel maintains a `vma server` and `page server` which work together to serve the missing pages for the follower OS and invalidate dirty pages (those replicated pages being written). Thus any updates on HeterSec protected process space are synchronized across machine boundaries. HeterSec uses a fast in-kernel message handling layer to send messages across nodes. Since it directly involves the kernel network drivers (e.g., RDMA over Gigabit Infiniband), the cost of switching between user-space and kernel-space is eliminated. Sending messages back and forth are relatively cheap between nodes. For example, the round-trip latency averages $17\mu\text{s}$ on RDMA in our micro-benchmark test, as described in Section 4.2.

We also implemented the system call RPC server to communicate over a fast message handling layer which similarly rides on RDMA over Infiniband. Similar to cross-node page and VMA handling, the master OS kernel registers a `system call server` in the message handling layer. The first time a process issues the `sys_hscall` system call (either an MTD or MVX), both master and follower kernels will mark that process as a *HeterSec process* (we introduce a flag in `task_struct`). For system calls that manipulate cross-node state (e.g., file, socket and event poll), the follower OS kernel verifies the file descriptor against the VDT to decide whether to invoke the remote system call handler in the master OS or execute them locally. Note that the follower kernel will only check system calls of HeterSec processes, any other processes will be free from this inspection.

To enable cross-ISA program state transformation, we leverage the open source Popcorn compiler [4-6] to embed all the ISA related metadata into the executable. Such information includes the ISA specific instructions, the state relocation mapping, as well as the ISA-switching points. The state relocation mapping is used at each ISA-switch point, with which a translation library transfers the currently running process state (e.g., register states, stack slots, etc.) from one ISA to another. The compiler was built on LLVM, and all the ISA specific code instrumentation was implemented as several

middle-end and backend passes. When compile applications into ISA-specific binaries, we use the same LLVM IR to generate the assembly code for each architecture. Therefore, the stack variables in different architecture can be mapped based on the same origin in IR.

4 Evaluation

In this section, we evaluate the HeterSec prototype as well as its two applications in terms of the security benefits and the performance overhead. All the experiments were evaluated on an x86_64 and ARM64 machine pair. The x86_64 server contains an Intel Xeon E5-2620v4 CPU with the clock speed of 2.1GHz. The ARM64 server contains a Cavium ThunderX CPU (ARMv8.1) with clock speed of 2.0GHz. The two machines are equipped with 32GB and 128GB of DRAM respectively, and they are connected using Mellanox ConnectX-4 InfiniBand with a bandwidth of up to 56 Gbps.

4.1 Security Analysis

Similar to existing diversification-based defense systems, HeterSec also leverages randomized and unknown target process address information (a.k.a ASLR) for the baseline security. However, heterogeneous-ISA based approaches could bring an additional layer of ISA diversity for the process, making it harder for attackers to generate payloads that fit both architectures. Similar to most of the existing diversification systems, we assume the attackers have remote access to the target process with a known interface (e.g., connection sockets). However, HeterSec provides a black box of ISA diversified instances to attackers. With HeterSec, we can leverage the ISA divergent hardware and compilation toolchain to generate program instances with differing instruction sets. The generated application instances also possess different calling conventions, variable register usages, and differential stack layouts. For example, ARM64 allows at most 8 general-purpose registers ($x0 - x7$) to be used for passing function call parameters; while x86_64 only has at most 6 general-purpose registers for passing parameters. In terms of the system call, ARM64 uses $x8$ register for system call number and $x0$ for system call return value; while x86_64 uses rax for both system call number and return value. Furthermore, most security essential system calls have different system call numbers in the two architectures (e.g., the system call number of `execve` is 59 on x86_64 and 221 on ARM64). This altogether brings extra difficulties for attackers to launch an attack by, for example, return oriented programming.

One observation is that stack operations behave differently on ARMv8 and x86_64. ARMv8 stores the frame pointer (FP) and the link register (LR) both on the lowest address of the stack frame. Whereas x86_64 pushes the instruction pointer (RIP) and the stack base pointer (RBP) into the highest address of the stack frame. The slight difference in control

pointer location will make it hard for most of the stack based control flow hijacks to work on both instances. To further prove that the ISA diversified instances will have differing memory layouts, we wrote a tool utilizing `ptrace` and `capstone` [65] to dump the code and data pointers of a running process. We examined the potential pointers in `.data`, `.stack`, and `.heap`, and found 7846 pointers in the x86_64 version of `lighttpd` while there were 10385 pointers in a `lighttpd` web server running on ARM64. Despite the large number of pointers found in each binary we only found 3 pointers which had overlapping addresses between the two `lighttpd` processes running on these different ISAs. In the above mentioned experiment, we only examined the pointers with their relative addresses from the base of code segment. That means in reality, there will be almost zero chance of overlapping pointers, since ASLR disturbs the base code addresses of those program instances [2]. In addition, we also examined some real-world exploits on HeterSec environment. One example is CVE-2013-2028, in which an integer overflow and a buffer overflow in the Nginx `ngx_http_read_discarded_request_body()` function are used to gain control over the execution flow and carry out ROP attacks [10]. To trigger the vulnerability an attacker first sends a HTTP chunked request with a large chunked length, resulting in a negative integer to be casted to an unsigned `size_t` type. Subsequently this causes a `recv` call to read in a value larger than the buffer size from the client, leading to a buffer overflow. We ran a ROP attack script leveraging the CVE-2013-2028 buffer overflow [74] and while it was able to execute and trigger the vulnerability on an x86_64 machine, the script failed on the ARM64 machine and caused the Nginx process to crash and restart. The stack layouts between architectures differ therefore the address at which the overflow gains control over the program control flow are not the same.

Another interesting benefit of a multi-ISA security system is that it could potentially raise the bar for micro-architecture attacks [37, 46]. This is due to the fact multiple attack primitives have to be implemented differently on different architectures. For example, cache timing measurement and cache flush have different implementation details. In terms of cache timing measurement, attackers could use an unprivileged `rdtsc` instruction on x86_64 hardware. However, the similar performance counter is only accessible in kernel space on both ARMv7 and ARMv8 processors. Similarly, attackers can directly flush the cache line with `clflush` on x86_64, but have to carefully construct a memory access footprint that defeats the cache replacement policy in order to flush the cache line on ARM processors [45]. The run-time cache layout and timing diversities increases the cost to launch such attacks. Besides the diversified instance memory layout and the micro-architecture behaviors, multi-ISA software diversification could also allow the protected process to hybridize the architecture specific features for increased security. For example, the protected process running on x86_64 can potentially

switch to ARMv8 and validate whether the pointers were modified by attackers with ARMv8 pointer authentication, while still making use of the Intel MPK or MPX hardware features to secure memory page accesses and check boundaries [43, 50, 54, 77].

4.2 Performance Evaluation

To evaluate the performance impact on multi-ISA security applications, we first report the costs involved in cross-node operations such as remote system calls and ISA-switches. Next, we evaluated HeterSec with real-world applications.

Micro-benchmark To get a breakdown of these costs, we implemented a simple micro-benchmark to execute code remotely on the follower node, which triggers 100,000 ISA-switches. We measured the network latency on an ARM64 machine node using a x86_64 node as the follower. As shown in Table 1, a remote system call like `getpid()` imposes an additional $\sim 17.6 \mu s$ overhead when being called compared with the native execution of the `getpid()` system call. The primary reason for this overhead is the unavoidable communication cost brought by the dual-node architecture. The result matches the raw network ping-pong micro-benchmark ($\sim 17.62 \mu s$), in which we wrote a simple kernel module sending 100,000 short messages back and forth between the two machines. Interestingly, this cross-node network latency is much smaller than the network latency observed using Linux `ping` command ($\sim 112 \mu s$). This is because the HeterSec messaging APIs are implemented in the kernel, thus it avoids the complicated TCP/IP network stack and the user/kernel context switch cost. We also observed the ISA switching cost ($\sim 504.85 \mu s$) in our micro-benchmark is higher than a remote system call, this is mainly caused by the cross-ISA program state transformation and the page synchronization.

Table 1: The cost (in μs) of remote system call and ISA-switch compared to local `getpid()` system call on x86_64.

Operations	Latency (in μs)
<code>getpid()</code>	0.47 ± 0.01
remote <code>getpid()</code>	18.09 ± 0.36
raw network ping-pong	17.62 ± 0.33
ping latency	112 ± 15
ISA-switch	504.85 ± 4.70

Application Benchmarks We selected the *nbench* benchmark suite [12], two web server applications – *Nginx* and *Lighttpd*, an in-memory database *Redis* server and a file compression utility *GNU gzip*. We used *nbench* to measure the performance of HeterSec on CPU and memory intensive workloads. *Nbench* is a compute, FPU and memory intensive benchmark suite containing some common computation

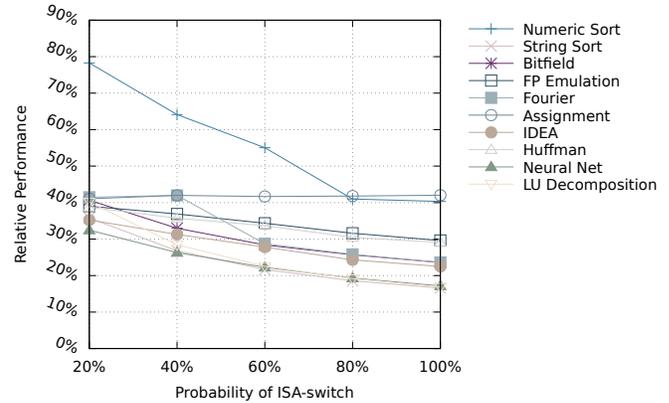


Figure 4: Performance of *nbench* with the probability of 20%, 40%, 60%, 80% and 100% to switch to the counterpart ISA respectively. The numbers are normalized with zero ISA switch, execution on the x86_64 node.

workloads, such as string sorting and neural network back propagation. We measured the performance overhead of using multi-ISA hardware under the security scenarios mentioned in Section 3.4 and reported the performance overhead incurred.

We first measured the performance impact of running *nbench* under variable probabilities to switch to the counterpart node with different ISA (the MTD scenario). In the experiment, we started the *nbench* program on the x86_64 node; the code will be executed randomly on each node afterwards. We measured the execution time of each test case and normalized to zero probability of ISA-switch (all code executed on x86_64 node). We show the normalized performance overhead numbers in y-axis of Figure 4. As expected, the performance decreases as the probability to perform an ISA-switch increases. This is because the ISA-switch is a relatively expensive operation; the higher chance of ISA-switches during application execution, the more overall performance overhead each application could have. When the first time program execution switches to the counterpart node, HeterSec kernels have to load the code and setup the kernel data structure. This explains the reason that some benchmarks lose 50% performance even under 20% chance of ISA-switch. Overall, execution transfer across nodes contributes mainly for the performance degradation. Since the *nbench* is CPU and memory intensive, any latency incurred during the execution will have significant impact on relative performance.

To prove the feasibility of HeterSec on real-world applications, we evaluated the performance impact of executing *Nginx* and *Redis* in HeterSec MTD mode. *Nginx* and *Redis* are applications which are used in various commodity systems and best reflect the types of overheads which would be seen when deploying HeterSec in the field. We used *ApacheBench* to generate the HTTP requests to the web servers and queried for a web page of 4 KB size for 1 million times. We first run *ApacheBench* on a laptop located in the same LAN of the

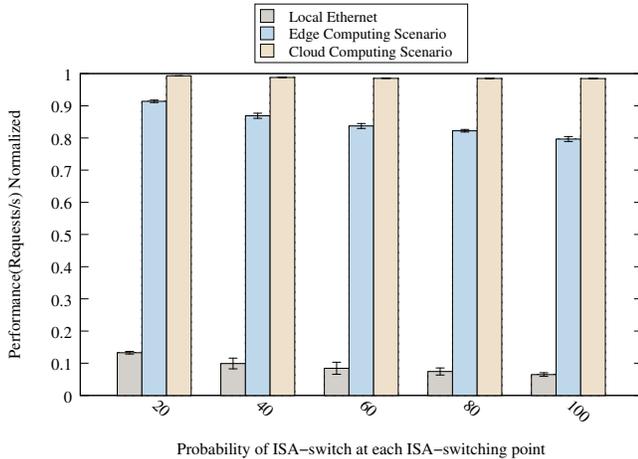


Figure 5: Performance of Nginx (requests/s) with variable probabilities to switch ISAs at every ISA-switching point.

target HeterSec machine pair. The laptop and the target machine pair are connected using a 10Gbps Ethernet with about 0.4 ms latency. We also run our test with artificial network latency of 10 ms and 40 ms respectively. The 10 ms latency is to emulate the typical latency seen in Edge Computing scenarios [14], while the 40 ms network latency could be seen as the minimal network overhead between two availability zones of the same region in the Amazon Web Services (AWS) cloud [66]. We manually configured the randomization probability at each ISA-switching point, and ran each test case 5 times. The average value and the standard deviations are reported in Figure 5. With a local network connection, Nginx on HeterSec performs at only about 11% throughput compared to the baseline. This is because internally an ISA-switch brings some additional costs of cross machine communication. Although the inter machine communication has been optimized by using fast in-kernel message API, the time spent on inter machine communication dominates the total request handing time. However, if we consider a real network scenario such as edge or cloud, HeterSec incurs a reasonable overhead. For example, we only observe a 10%-20% performance overhead depending on the frequency we trigger the migrations under 10 ms network latency (the edge computing scenario in Figure 5). At 100% ISA switch probability this equates to 5 switches per request, or about 3800 ISA switches per second. When testing on network designed to emulate the cloud (40 ms latency), the throughput of Nginx shows a very small drop in performance even with a 100% probability to switch ISAs (the cloud scenario in Figure 5). Note that HeterSec kernel brings minimal performance overhead to non-HeterSec processes. For example, the vanilla Nginx performs 22357.5 req/s on vanilla Linux kernel, whereas it performs 22273.8 req/s on HeterSec kernel (~0.37% overhead).

We observed similar results when running redis-benchmark to measure the throughput of Redis SET instructions. As

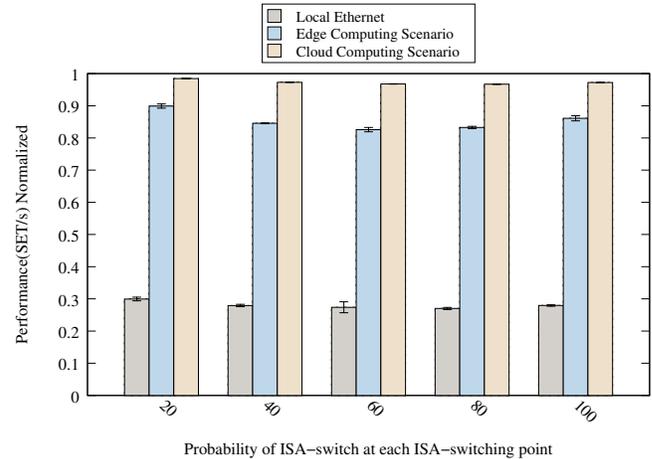


Figure 6: Performance of Redis (SET instructions/s) with variable probabilities to switch ISAs at every ISA-switching point.

shown in Figure 6, Redis performs about 30% throughput when running on HeterSec compared with the native execution. However, the overhead drops to 15% and 2% when running the benchmark over edge and cloud computing cases respectively. We set the ISA switch points in a periodic job for the Redis evaluation which resulted in about 20 ISA switches per second, pegged to the `server.hz` value. Interestingly enough, we saw a slight throughput improvement when we increased the ISA-switching probability threshold from 80% to 100%. This is likely due to the deterministic execution flow transfer avoiding destroying the branch prediction. The results show that although the frequent ISA-switch is expensive, it is feasible to use for server applications in real-world scenarios.

Next, we report the performance of two heterogeneous ISA multi-version execution prototypes. As mentioned in Section 3.4.2, the `ptrace` version multi-ISA MVX prototype is used to find out all the necessary system calls for simulation, as it is easier to debug with an userspace MVX engine. The MVX engine running in HeterSec kernels can achieve better performance. In our experiment, both MVX prototypes use the ARM64 node to launch the master variant, and offload the follower variant to the x86_64 node. The cost of MVX are mostly from the program state synchronization in between the two variants. For example, the master variant has to wait the system call simulation to be finished on the follower side in order to continue the execution (a.k.a., lock-step check).

We evaluated the two MVX prototypes with `nbench`, `gzip` and `Lighttpd` web server. `Gzip` and `Lighttpd` are two I/O intensive applications. In `gzip` test case, we randomly generate files in different size from `/dev/urandom`. We also used `ApacheBench` to generate workloads for `Lighttpd` web server. We run all the benchmarks with both kernel-based MVX and `ptrace`-based MVX prototypes. Figure 7 shows the normalized performance evaluation results using the vanilla application

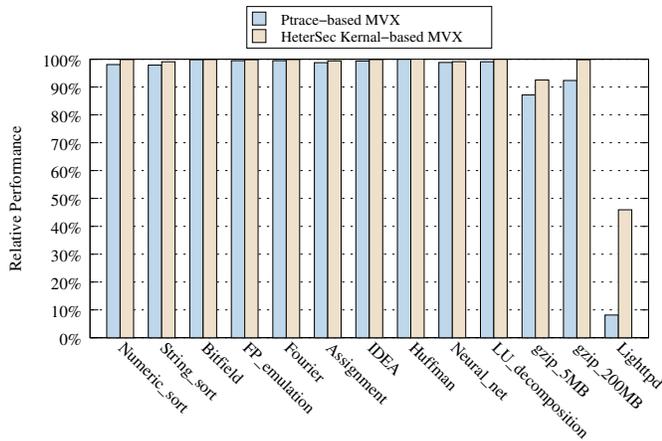


Figure 7: Relative performance of nbench, gzip, and Lighttpd running on the HeterSec kernel-based MVX and the ptrace-based MVX.

running on the ARM64 node as the baseline. For most of the CPU and memory-intensive workloads, kernel-based MVX and ptrace-based MVX have similar performance overheads. This is because most of the system calls in computation-intensive applications do not need to be simulated in the MVX engine. For I/O-intensive applications, both MVX engines process and check on descriptor related system calls such as `read/write(v)`. Overall, both multi-ISA MVX engines introduce about 10% overhead for the `gzip` benchmark. Since we duplicated the files on both nodes, there is no need to transfer data between nodes. For the web server application, the MVX engines have to simulate a number of network I/O related system calls, including `accept4`, `socket`, `sendfile` and `recvfrom`, etc. In general, the HeterSec kernel-based MVX engine pulls down the `Lighttpd` throughput to about 50% of its native performance. However, that performance is still better than the ptrace-based MVX engine ($\sim 10x$) and the MVX engine in DMON ($\sim 5.43x$) [73].

5 Related Works

The first category of related work is the *various techniques for software diversity* [42]. An important assumption for a software attack is the attacker could have the information of the target system [16, 59, 60], or at least by chance to obtain such information by, for example, brute forcing [10, 59]. It makes attacks easier if the code itself and the defense mechanisms are static. Software diversity provides uncertainty for the target system, which breaks the static nature of the target and thus increases the cost of an attack. For example, one of the notable software diversification techniques is ASLR (for most cases, in the form of code randomization) [3, 8, 13, 26, 36, 63, 78, 81]. Previous research demonstrated the effectiveness of code randomization at program module level [63], page level [3], func-

tion level [36], basic block level [13, 78], or even instruction level [26]. Some latest research further show the feasibility of ASLR at runtime, making the code layout re-randomized for a given period of time [8, 13, 81]. HeterSec extends this research line by exploring the feasibility of using heterogeneous instruction set to diversify the program.

Multi-version execution is another concrete technique of software diversity. Instead of randomizing a single code instance, MVX engines run multiple variants of program instances simultaneously [15, 38, 51, 57, 58, 72, 73, 83]. Those variants are different in memory layout, so that a malicious input might trigger the vulnerable code in one variant but likely to fail on other variants. Such memory layout differences could be non-overlapping memory map [38, 57, 83], reverse stack growth [58], etc. Recently, researchers also proposed to apply MVX inside Linux kernel, to detect kernel bug exploits [83]. DMON is a very recent and concurrent work using distributed heterogeneous-ISA machines for multi-version execution [73]. DMON shows that MVX with heterogeneous ISA setting can achieve better effectiveness for advanced code reuse attacks, such as the position-independent ROP [23, 73]. As we have compared in Section 2.2, DMON focuses on a heterogeneous-ISA MVX engine only, whereas HeterSec is proposed as a general framework. The multi-ISA MVX engine is a showcase of the HeterSec application scenarios.

Another category of the related work includes the split-interface systems [18, 19, 62, 75] and the multikernel OSes [4, 6, 7, 79]. The split-interface systems normally leverage two compartments to separate and isolate program code execution or secret data access. For example, `proxos` [62] splits the application execution into trusted and untrusted parts. The trusted part of the execution is isolated in a separate private VM, while the untrusted code can only communicate with the trusted code through a proxy OS. Nested kernel [18] and `SecPod` [75] split the OS kernels into isolated components for enhanced kernel security. `Isomeron` [19] on the other hand splits the code execution between two diversified variants. By randomly “flip-coin” selecting the next function to be executed, `Isomeron` randomizes the execution path to mitigate conventional code reuse attacks [19]. HeterSec shares the same idea of splitting interface to secure application execution, but HeterSec further enhances the execution security by split-executing code on two ISA-diversified nodes.

The multikernel OS treats a multi-core machine as a distributed network of independent cores. A number of systems leverage multiple OS kernels to manage the heterogeneous and multi-core machines in a divide and conquer way [4, 6, 7, 79]. For example, `Barrelfish` [7] runs multiple OS kernels on top of a multi-core machine in order to make multi-thread application performance scalable. Similarly, `fos` tackles the scalability issues by factoring the OS into micro-kernel components [79]. `Popcorn Linux` is a most similar work that runs multikernel on heterogeneous hardware [4, 6]. `Popcorn Linux` focuses on single-threaded HPC applications migra-

tion; on the other hand, HeterSec targets a not well explored research area – the feasibility of securing an application execution with ISA diversity. Furthermore, server applications and multi-threaded applications are supported with HeterSec.

6 Conclusion

In this paper, we explored the missed research space of securing application execution with ISA diversity. We described the design and implementation of HeterSec, a framework to improve application security with ISA heterogeneity. HeterSec enables HeterSec processes to leverage the diversified ISAs as an additional layer of dynamic defense. HeterSec was built with several compiler and kernel extensions to facilitate processes running on heterogeneous hardware in a security enhanced manner. The two security applications built on HeterSec show that it is feasible to leverage the existing heterogeneous hardware to improve application security.

The source code of HeterSec is publicly available as part of the Popcorn Linux project at <http://popcornlinux.org>.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is supported in part by grants received by Virginia Tech including that from the US Office of Naval Research (ONR) under grants N00014-18-1-2022, N00014-16-1-2104, and N00014-16-1-2711, and from NAVSEA/NEEC under grant N00174-16-C-0018. Kim’s work at Virginia Tech (former affiliation) was supported by ONR under grants N00014-16-1-2711 and N00014-18-1-2022. Olivier’s work at Virginia Tech (former affiliation) was supported by ONR under grants N00014-16-1-2104 and N00014-18-1-2022. Lyerly’s work at Virginia Tech (former affiliation) was supported in part by NAVSEA/NEEC under grant N00174-16-C-0018.

This work is also supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1310).

References

- [1] ARM Limited (or its affiliates). ARM BIG.LITTLE. <https://www.arm.com/why-arm/technologies/big-little>, Accessed: 2020-07-08.
- [2] Linux Kernel Address Space Layout Randomization. <http://lwn.net/Articles/569635/>.
- [3] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. *Proc. 23rd Usenix Security Sym*, pages 433–447, 2014.
- [4] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, volume 52, pages 645–659. ACM, 2017.
- [5] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel os based on linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.
- [6] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys ’15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [7] Baumann, Andrew and Barham, Paul and Dagand, Pierre-Evariste and Harris, Tim and Isaacs, Rebecca and Peter, Simon and Roscoe, Timothy and Schüpbach, Adrian and Singhanian, Akhilesh. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.
- [8] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.
- [9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
- [10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking Blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 227–242. IEEE, 2014.
- [11] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.
- [12] BYTEmark benchmark. Linux/Unix nbench. <http://www.math.utah.edu/~mayer/linux/bmark.html>, Accessed: 2020-07-08.
- [13] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.

- [14] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proceedings of the 11th annual workshop on network and systems support for games*, page 2. IEEE Press, 2012.
- [15] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, 2006.
- [16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.
- [17] Daniel, P and Marco, Cesati and others. Understanding the Linux kernel, 2007.
- [18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.
- [19] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Iso-meron: Code Randomization Resilient to (just-in-time) Return-oriented Programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)*, 2015.
- [20] Matthew DeVuyst, Ashish Venkat, and Dean M Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 261–272. ACM, 2012.
- [21] Hadi Esmailzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.
- [22] Peter N Glaskowsky. NVIDIA’s Fermi: the first complete GPU computing architecture. *White paper*, 18, 2009.
- [23] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 227–242. IEEE, 2018.
- [24] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moon-sang Kwon, Chanhoo Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA ’16*, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.
- [25] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.
- [26] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where’d My Gadgets Go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585. IEEE, 2012.
- [27] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.
- [28] Intel. *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, Nov 2019. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf>.
- [29] Intel. Intel Xeon processor scalable family. <https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html>, Accessed: 2020-07-08.
- [30] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Open-flow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012.
- [31] Jajodia, Sushil and Ghosh, Anup K and Swarup, Vipin and Wang, Cliff and Wang, X Sean. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, volume 54. Springer Science & Business Media, 2011.
- [32] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.
- [33] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates,

- Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datascenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.
- [34] Shubham Kamdar and Neha Kamdar. big.LITTLE architecture: Heterogeneous multicore processing. *International Journal of Computer Applications*, 119(1), 2015.
- [35] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.
- [36] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-grained Randomization of Commodity Software. In *Computer Security Applications Conference, 2006. ACSAC’06. 22nd Annual*, pages 339–348. IEEE, 2006.
- [37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.
- [38] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and Efficient Multi-Variant Execution using Hardware-Assisted Process Virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.
- [39] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.
- [40] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477, 2018.
- [41] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [42] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP ’14, 2014.
- [43] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, J Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. *arXiv preprint arXiv:1811.09189*, 2018.
- [44] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14*, pages 285–300, New York, NY, USA, 2014. ACM.
- [45] Moritz Lipp. *Cache attacks on arm*. PhD thesis.
- [46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Melt-down: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.
- [47] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP ’11*, page 327–336, New York, NY, USA, 2011. Association for Computing Machinery.
- [48] Timothy P. Morgan. Tiler rescues CPU cycles with network coprocessors, 2013. <https://bit.ly/2DfM53R>.
- [49] Netronome. Agilio SmartNICs, 2019. <https://www.netronome.com/products/smartnic/overview/>.
- [50] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28, 2018.
- [51] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. Multi-variant execution atop a decomposed hypervisor on emerging heterogeneous-isa multicore. 2016. EuroSys’16 (Poster).
- [52] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. OS Support for Thread Migration and Distribution in the Fully Heterogeneous Datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 174–179. ACM, 2017.
- [53] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Dynamos: dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 322–333. ACM, 2015.

- [54] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.
- [55] QEMU. <http://www.qemu.org>.
- [56] Stefan Rusu, Simon Tam, Harry Muljono, Jason Stinson, David Ayers, Jonathan Chang, Raj Varada, Matt Ratta, Sailesh Kottapalli, and Sujal Vora. A 45 nm 8-core enterprise Xeon processor. *IEEE Journal of Solid-State Circuits*, 45(1):7–14, 2010.
- [57] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.
- [58] Salamat, Babak and Gal, Andreas and Franz, Michael. Reverse stack execution in a multi-variant execution environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, pages 1–7, 2008.
- [59] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04*, pages 298–307, 2004.
- [60] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.
- [61] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation Intel Xeon Phi. *IEEE micro*, 36(2):34–46, 2016.
- [62] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association.
- [63] PaX Team. PaX Address Space Layout Randomization (ASLR), 2003.
- [64] Marvell Technology. Liquidio ii 10/25gbe Adapter family, 2019. <https://bit.ly/2H7NWLk>.
- [65] The Ultimate Disassembly Framework – Capstone. <http://www.capstone-engine.org/>.
- [66] AWS Inter-Region Latency. <https://www.cloudping.co/>.
- [67] Reduce TCO with Arm Based SmartNICs. <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/reduce-tco-with-arm-based-smartnics>.
- [68] High-Performance Programmable SmartNICs. <https://www.mellanox.com/products/smartnic/>.
- [69] Ashish Venkat, Harsha Basavaraj, and Dean Tullsen. Composite-isa cores: Enabling multi-isa heterogeneity using a single isa. In *25th IEEE International Symposium on High Performance Computer Architecture*. IEEE, February 2019.
- [70] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 727–741. ACM, 2016.
- [71] Ashish Venkat and Dean M Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. *ACM SIGARCH Computer Architecture News*, 42(3):121–132, 2014.
- [72] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450, 2016.
- [73] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. Dmon: A distributed heterogeneous n-variant system. *arXiv preprint arXiv:1903.03643*, 2019.
- [74] w00d. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028). <https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html>.
- [75] Xiaoguang Wang, Yong Qi, Zhi Wang, Yue Chen, and Yajin Zhou. Design and Implementation of SecPod, A Framework for Virtualization-Based Security Systems. *IEEE Transactions on Dependable and Secure Computing*, 16(1):44–57, 2019.
- [76] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Sang-Hoon Kim, and Binoy Ravindran. A Framework to Secure Applications with ISA Heterogeneity. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA)*, 2019.

- [77] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.
- [78] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.
- [79] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.
- [80] Wikipedia. MSI Protocol. https://en.wikipedia.org/wiki/MSI_protocol, Accessed: 2020-07-08.
- [81] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*, pages 367–382, 2016.
- [82] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, volume 1, pages 22–25, 2014.
- [83] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *ASPLOS*, April 2019.

Confine: Automated System Call Policy Generation for Container Attack Surface Reduction

Seyedhamed Ghavamnia
Stony Brook University

Tapti Palit
Stony Brook University

Azzedine Benameur
Cloudhawk.io

Michalis Polychronakis
Stony Brook University

Abstract

Reducing the attack surface of the OS kernel is a promising defense-in-depth approach for mitigating the fragile isolation guarantees of container environments. In contrast to hypervisor-based systems, malicious containers can exploit vulnerabilities in the underlying kernel to fully compromise the host and all other containers running on it. Previous container attack surface reduction efforts have relied on dynamic analysis and training using realistic workloads to limit the set of system calls exposed to containers. These approaches, however, do not capture exhaustively all the code that can potentially be needed by future workloads or rare runtime conditions, and are thus not appropriate as a generic solution.

Aiming to provide a practical solution for the protection of arbitrary containers, in this paper we present a generic approach for the automated generation of restrictive system call policies for Docker containers. Our system, named *Confine*, uses static code analysis to inspect the containerized application and all its dependencies, identify the superset of system calls required for the correct operation of the container, and generate a corresponding Seccomp system call policy that can be readily enforced while loading the container. The results of our experimental evaluation with 150 publicly available Docker images show that *Confine* can successfully reduce their attack surface by disabling 145 or more system calls (out of 326) for more than half of the containers, which neutralizes 51 previously disclosed kernel vulnerabilities.

1 Introduction

The convenience of running containers and managing them through orchestrators, such as Kubernetes [13], has popularized their use by developers and organizations, as they provide both lower cost and increased flexibility. In contrast to virtual machines, which run their own operating system (OS), multiple tenants can launch containers on top of the same OS kernel of the host. This makes containers more lightweight compared to VMs, and thus allows for running a higher number of instances on the same hardware [30].

The performance gains of containers, however, come to the expense of weaker isolation compared to VMs. Isolation between containers running on the same host is enforced purely in software by the underlying OS kernel. Therefore, adversaries who have access to a container on a third-party host can exploit kernel vulnerabilities to escalate their privileges and fully compromise the host (and all the other containers running on it).

The trusted computing base in container environments essentially comprises the entire kernel, and thus all its entry points become part of the attack surface exposed to potentially malicious containers. Despite the use of strict software isolation mechanisms provided by the OS, such as capabilities [1] and namespaces [18], a malicious tenant can leverage kernel vulnerabilities to bypass them. For example, a vulnerability in the `waitid` system call [6] allowed malicious users to run a privilege escalation attack [70] and escape the container to gain access to the host.

At the same time, the code base of the Linux kernel has been expanding to support new features, protocols, and hardware. The increase in the number of exposed system calls throughout the years is indicative of the kernel's code "bloat." The first version of the Linux kernel (released in 1991) had just 126 system calls, whereas version 4.15.0-76 (released in 2018) supports 326 system calls. As shown in previous works [40, 50, 51, 80], different applications use disparate kernel features, leaving the rest unused—and available to be exploited by attackers. Kurmus et al. [50] showed that each new kernel function is an entry point to accessing a large part of the whole kernel code, which leads to attack surface expansion.

As a countermeasure to the ever expanding code base of modern software, *attack surface reduction* techniques have recently started gaining traction. The main idea behind these techniques is to identify and remove (or neutralize) code which, although is part of the program, it is either i) completely inaccessible (e.g., non-imported functions from shared libraries), or ii) not needed for a given workload or configuration. A wide range of previous works have applied this concept at different levels, including removing unused functions from shared libraries [56, 58, 66] or even removing

whole unneeded libraries [47]; tailoring kernel code based on application requirements [50, 80]; or limiting system calls for containers [8, 68, 69, 75]. In fact, one of the suggestions in the NIST container security guidelines [59] is to reduce the attack surface by limiting the functionality available to containers.

Despite their diverse nature, a common underlying challenge shared by all these approaches is how to accurately identify and maximize the code that can be *safely* removed. On one end of the spectrum, works based on static code analysis follow a more conservative approach, and opt for maintaining compatibility in the expense of not removing all the code that is actually unneeded (i.e., “remove what is not needed”). In contrast, some works rely on dynamic analysis and training [8, 50, 68, 69, 75, 80] to exercise the system using realistic workloads, and identify the actual code that was executed while discarding the rest (i.e., “keep what is needed”). For a given workload, this approach maximizes the code that can be removed, but as we show in Section 4, it does not capture exhaustively *all* the code that can *potentially* be needed by different workloads—let alone parts of code that are executed rarely, such as error handling routines.

Given that previous efforts in the area of attack surface reduction for container environments have focused on dynamic analysis [8, 68, 69, 75], in this work we aim to provide a more generic and practical solution that can be readily applied for the protection of any container without the need for training. To that end, we present an automated technique for generating restrictive system call policies for arbitrary containers, and limiting the exposed interface of the underlying kernel that can be abused. By relying on static code analysis, our approach inspects *all* execution paths of the containerized application and all its dependencies, and identifies the superset of system calls required for the correct operation of the container.

Our fully automated system, named *Confine*, takes a container image as its input and generates a customized system call policy. Containers, once initialized, run a single application for their entire execution time. We use dynamic analysis to capture all binary executables that might be invoked during container initialization. This initial limited dynamic analysis phase does not depend on the availability of any workloads, and just pinpoints the set of executables that are invoked in the container, which are then statically analyzed. We have chosen Docker as the main supported type of container images, as it is the most widely used open-source containerization technology.

We experimentally evaluated our prototype with a set of 150 publicly available Docker images, and demonstrate its effectiveness in deriving strict system call policies without breaking functionality. In particular, for about half of the containers, Confine disables 145 or more system calls (out of 326), while at least 100 or more system calls are disabled in the worst case and 219 in the best case. This is in stark contrast to Docker’s default list of 49 (plus four partially) disabled system calls. More importantly, disabling these system calls effectively *neutralizes 51 previously disclosed*

kernel vulnerabilities, in addition to the 25 vulnerabilities mitigated by Docker’s default Seccomp policy.

The main contributions of our work include:

- We propose a generic approach for the automated generation of restrictive, ready-to-use Seccomp system call policies for arbitrary containers, without depending on the availability of source code for the majority of the target programs.
- We performed a thorough analysis of Linux kernel CVEs, mapping them to functions in the kernel code. We identified which system calls can be used to exploit each CVE, and used this mapping as the basis for evaluating the effectiveness of our approach.
- We examined more than 200 of the most popular publicly available Docker images from Docker Hub [7] and present an analysis of their characteristics.
- We experimentally evaluated our system with the above images and demonstrate its effectiveness in generating restrictive system call policies, which neutralize 51 previously disclosed kernel vulnerabilities.

Our Confine prototype is publicly available as an open-source project from <https://github.com/shamedgh/confine>.

2 Background

The attack surface of the OS kernel used by containers can be reduced by *restricting* the set of system calls available to each container. In this section, we describe how Linux containers provide isolation to different “containerized” processes, and how SECure COMPUting with filters (Seccomp BPF) [23] can be used to reduce the kernel code exposed to containers.

2.1 Linux Containers

Linux containers are an OS-level virtualization approach, which can be used to execute multiple userlands on top of the same kernel. The Linux kernel uses Capabilities [1], Namespaces [18] and Control Groups (cgroups) [3] to provide isolation among different containers.

Namespaces are a kernel feature that virtualizes *global* system resources (specifically: mount points, process IDs, network devices and network stacks, IPC objects, hostnames, user and group IDs, and cgroups), providing the “illusion” of exclusive use of these resources to processes within the same namespace. Control Groups allow processes to be organized into hierarchical groups, whose usage of various types of resources (e.g., CPU time, memory, disk space, disk and network I/O) can be limited, accounted, or prioritized accordingly. Containers use cgroups to provide “fair” usage of resources.

Docker [7] is a platform that employs the software-as-a-service and platform-as-a-service models for developing,

deploying, and running containers. Every Docker container launched is based on a Docker image, which is a file built in layers, encapsulating the entire environment (including a whole Linux distribution, libraries, and support utilities) required to execute the containerized application(s). The specification of the Docker image is described in a text file, called *Dockerfile*. The Dockerfile essentially contains all the commands required to assemble the respective image. Docker uses Linux namespaces and cgroups to provide isolation between containers.

Docker Hub [7] is a central repository of both community-based and official Docker images, which has drastically popularized container use among system administrators. More importantly, by building streamlined services with a minimal code base, Docker has enabled corporations to increasingly switch to the use of microservices. Each microservice can be configured as a Docker image once, and then multiple instances of it can be launched.

2.2 Seccomp BPF

User-space applications communicate with the OS kernel through the provided set of *system calls*, i.e., a pre-defined API that allows access to specific kernel functionalities programmatically. More importantly, however, applications typically need only a *subset* of the available system calls to function properly, i.e., most applications do not make use of all the provided system calls.

Nevertheless, although a program may not use all of the provided system calls, the complete set is available to all processes. This modus operandi has two issues: (1) a compromised application may use additional system calls (from what the author of the application originally intended) to carry out malicious operations that require access to system resources (e.g., filesystem, network) that the application never meant to access; and (2) a malicious (or compromised) application may invoke unused system calls to exploit underlying kernel vulnerabilities (typically related to the implementation of a given system call) for privilege escalation [45, 46], thereby gaining access to every process and container on the host.

Seccomp BPF [23] is a mechanism for restricting the set of system calls that are accessible by a given application. Specifically, Seccomp BPF uses the Berkeley Packet Filter language [55] for allowing developers to write arbitrary programs that act as system call filters, i.e., BPF programs that inspect the system call number (as well as argument values, if needed) and allow, log, or deny the execution of the respective system call. Docker containers can be executed with Seccomp BPF profiles, allowing users to provide allow/deny lists of permitted/prohibited system calls. The specified allow/deny list is applied to the entire process namespace, limiting all processes executed inside the respective container. We use this mechanism to reduce the kernel code available to each container.

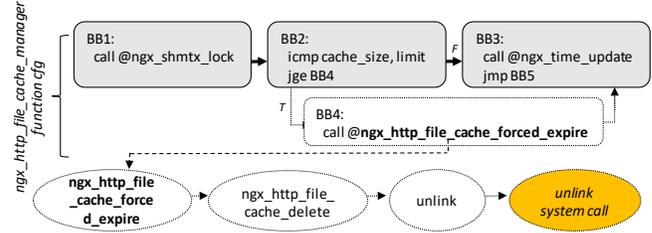


Figure 1: Example of control flow in Nginx that is missed by dynamic analysis. Ovals represent functions, while rectangles represent basic blocks. Dashed branches and blocks are not executed during training.

3 Threat Model

We consider a local adversary who has full access to a container running on a third-party host. This access may be granted either legitimately (e.g., as a regular user of a cloud service), or as a result of compromising a vulnerable process running on the container. Potential victims include the OS kernel of the host, as well as any other containers running on it. We specifically focus on preventing the attacker from escaping a container—preventing the exploitation of an application running on a container is not the focus of our work. Any exploit mitigations and defenses deployed on the host or individual containers are orthogonal to our approach, as it does not rely on any additional protection mechanisms being in place at user or kernel space.

Confine limits the set of system calls an attacker can invoke. In case of vulnerability exploitation, this means that exploit code (e.g., shellcode or ROP payload) or malicious programs run by the attacker will have more limited capabilities, as they cannot rely on system calls that are not needed by the container. More importantly, by preventing access to less frequently used and less tested system calls—the kernel code of which may contain vulnerabilities that can lead to privilege escalation [53]—an attacker cannot trigger those vulnerabilities to compromise the kernel, as the respective system calls cannot be invoked in the first place.

4 The Need for Static Analysis

Previous works [8, 68, 69, 75] have used dynamic analysis to derive the list of system calls used by a container. However, dynamic analysis is not sound, and thus can miss system calls along execution paths that were not exercised during the training phase. To demonstrate this issue, we manually analyzed Nginx and discovered three examples of system calls that would be missed if only dynamic analysis were used. For our evaluation, we use Nginx with the Cache Management and Auto Index features enabled.

Nginx spawns a separate *cache-manager* process to handle cache management. This process clears the older cached files when the cache is full using the `unlink` system call. Dynamically analyzing Nginx would capture the initialization

of the cache-manager process, but would likely fail to capture the *deletion* of older cached files, and therefore fail to capture the use of the `unlink` system call. As the `unlink` system call is not invoked anywhere else during the normal execution of the program, relying on training alone would cause it to be marked as unused. Moreover, extending the training phase for a longer duration would not solve the problem because the deletion of older files is triggered only when the cache is full. Training would need to request enough *new* files to fill up the cache. Correctly setting up the training process to handle such situations is thus challenging. Figure 1 shows the parts of the control flow that are not discovered during training.

Another example of failure to capture a system call is the use of `lstat` when displaying directory listings. Apart from this functionality, `lstat` is not used in any other part of Nginx. As listing a directory is usually triggered by users who manually type a URL, and not by following any existing URL on a website, it is unlikely that a training-based approach would be able to capture this system call.

In yet another case, the Nginx binary can be updated with a newer version without dropping client connections. The system calls `getsockopt` and `getsockname` are used to hand over the existing socket connections to the new process, and are not used anywhere else in the code, making it challenging for dynamic analysis to discover them.

The above examples are indicative of the trade off between fragility and overapproximation faced by dynamic and static analysis. Relying on dynamic analysis alone would require the training to be comprehensive enough to anticipate and capture all above corner cases. In contrast, static analysis results are guaranteed to be sound, but may include system calls that are never invoked by certain workloads. As we aim for a practical and generic solution, we opt for using static analysis to capture the superset of system calls used by an application.

5 Design

Our goal is to reduce the kernel attack surface available to a malicious tenant of a container service by limiting the number of system calls available to each container, which can potentially be of use for malicious purposes (either as part of exploit code, or as a gateway to exploiting kernel vulnerabilities). To achieve this, Confine “hardens” the container image once it has been fully configured by the user, by limiting access to only those system calls that are actually needed for the proper operation of the container.

Identifying the system calls that are necessary for the correct execution of the container requires addressing the following requirements: 1) identify all applications that may run on the container; 2) identify all library functions imported by each application; 3) map library functions to system calls; and 4) extract direct system call invocations from applications and libraries.

Figure 2 presents a high-level overview of our approach, which, given a container image, automatically generates

Seccomp rules that limit the system calls that may be invoked. Confine currently supports Docker containers running on a native Linux-based host, but similar analysis could be performed for other container environments and operating systems.

5.1 Identifying Running Applications

Although containers are usually specialized to run a single application or service, they typically invoke many other utility and support programs prior to executing the main program. For example, the default MongoDB Docker image [16] invokes the following supporting programs to set up the environment: `bash`, `chown`, `find`, `id`, and `numactl`. To generate system call policies, we must thus identify all programs that can potentially run during the lifetime of a container. Confine relies on limited dynamic analysis to capture the list of processes created on the system. A profiling tool records every application launched within a configurable time period (30 seconds by default) since the creation of the container—long enough to capture both system initialization, as well as the “stable” state of the system. The obtained set of applications is then used to derive the corresponding system call policy. We further discuss the completeness of the derived list in Section 8.

Our approach is different from previous works that rely on dynamic training using various workloads to derive a list of allowable system calls [75]. In our approach, the goal of the dynamic analysis is merely to identify the set of binary executables to be analyzed—the system calls invoked by these programs are then derived statically.

The above dynamic analysis is meant to be a convenient and automated way to carry out the batch analysis of multiple container images. For containers that may include applications that are not launched from the beginning, our system supports manually provided external lists of executables that should be included in the analysis.

5.2 Static Analysis

Dynamic analysis often fails to exercise all possible code paths, especially when comprehensive workloads are not available during training. To ensure complete code coverage, once we have the list of applications that are executed on the container, we perform static analysis to extract the system calls that are needed for the correct execution of each application.

Libc User programs typically invoke system calls through the `libc` library, which provides corresponding wrapper functions (e.g., the `libc` function `read` invokes the system call `SYS_read`). Confine analyzes the source code of `libc` to derive a mapping between exported functions and the system calls they invoke. For the rest of the programs and libraries on a given container, however, Confine only needs to analyze their *binaries*.

A `libc` function may have multiple control flow paths to the actual system call. To correctly identify which system calls are

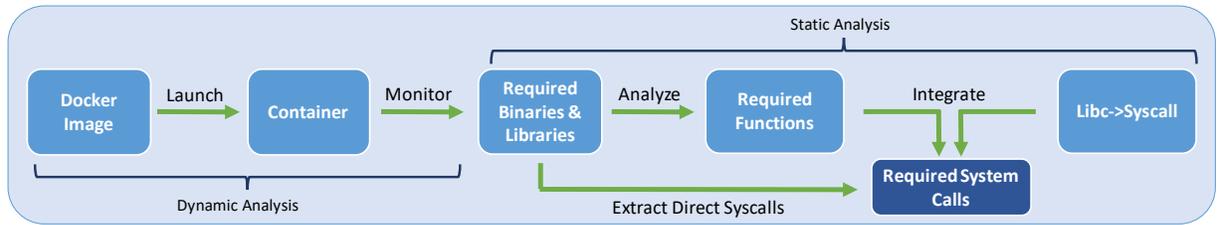


Figure 2: Overview of Confine’s system call extraction process. A one-time dynamic analysis phase that does not require any application-specific workloads is used for the sole purpose of identifying the applications running in the container. Each application is then statically analyzed to identify all the library functions that it uses, and the system calls it relies on.

invoked by a given libc function, we thus need to analyze these control flow paths. To that end, Confine statically analyzes the source code of libc to derive its full call graph, and accurately map each function to its respective system calls.

Function pointers are used widely in libc. However, performing accurate points-to analysis has significant scalability and performance issues [29, 42]. To avoid having to perform points-to analysis, we follow a more conservative approach and retain all system calls that are invoked through any function that has its address taken. In Section 6.1 we discuss the technical challenges we encountered during this process.

Having an accurate mapping between libc functions and system calls, it is then straightforward to analyze each program (main executable and libraries), identify all imported libc functions, and derive the set of all possible system calls the program may invoke. It is important to stress that this phase is performed only *once* per libc *version*—the derived mapping is then saved and used across all containers.

Direct System Call Invocation In addition to using libc wrappers, applications and libraries may also invoke system calls directly using the `syscall()` function, or using the `syscall` assembly instruction. Although the number of applications and libraries which use this approach are limited, for the sake of completeness, we use binary code disassembly to extract any directly invoked system calls. We describe in detail this process in Section 6.2. Some applications developed in languages other than C/C++ also require special considerations which we discuss in Section 6.3.

5.3 Hardening the Container Image

Once we have generated the list of system calls needed to run the container, we can proceed to harden the container image. Docker containers support the use of Seccomp filters to limit the system calls accessible from the container. The user can launch the container with a custom ruleset which specifies the system calls that can be accessed by the container. This ruleset can be either in the form of a deny list or an allow list of system calls prohibited or permitted. For Confine, we use a deny list of system calls that the container is not allowed to invoke.

Based on the analysis performed in Sections 5.1 and 5.2, we use an automated script to derive the list of prohibited system calls, and construct the corresponding Seccomp profile. If any new application needs to be executed on the container after this process, the administrator must run the analysis on the application to update the Seccomp profile.

6 Implementation

6.1 Mapping Libc Functions to System Calls

To ensure correctness, a precise function call graph is required to identify and filter unused system calls. Based on our analysis of more than 200 popular Docker images from Docker Hub [7], we found that even though most containers use the popular glibc library as their main user-space libc library, musl-libc [17] was also used in 12 occasions. Although both musl-libc and glibc provide implementations of the C standard library functions, and applications should be able to use both interchangeably, we discovered that the system calls used by standard libc functions some times differ between musl-libc and glibc.

To maximize compatibility, we analyzed both libraries independently to extract their call graphs and their corresponding function-to-system-call mapping. Moreover, due to certain differences between glibc and musl-libc, which we discuss next, we had to use a different toolchain for the analysis of each of these libraries.

6.1.1 Musl-Libc

Musl-libc [17] is a lightweight C standard library which has a smaller codebase compared to glibc. For our analysis, we compiled musl-libc with the LLVM [14] compiler toolchain and implemented an LLVM pass to extract the complete call graph. This pass operates on the intermediate representation (IR) of the code and records each function call. To identify system calls, in addition to recording each function call, we make special note of calls to the `syscall` function. Using the extracted call graph, we create a map between each exported function in musl-libc and the system calls it invokes. We modified the compiler toolchain to invoke the pass before

any optimization to prevent the loss of precision due to optimizations and code transformations.

Musl-libc uses a `weak_alias` macro to define weak symbols for functions. Weak symbols can be overridden by strong symbols having the same name, without name collision errors. Our LLVM pass keeps track of these aliases as well.

6.1.2 Glibc

Glibc is the most popular libc implementation used in most containers. Glibc heavily relies on multiple GCC [11] features which are not implemented in LLVM. Due to this issue, we implemented a second analysis pass to extract the call graph and system call information from glibc, based on the GCC RTL (Register Translation Language) Intermediate Representation. Our call graph extraction implementation is based on the Egypt [38] tool, which operates on GCC's RTL IR. We discovered that there are three main mechanisms through which glibc invokes system calls.

System Call via Inline Assembly and Assembly Files

This is the most straightforward mechanism for invoking system calls. Functions such as `accept4()`, which is responsible for accepting incoming socket connections, contain inline invocations using the x86-64 `syscall` instruction. Given the source code, the Egypt tool constructs the function call graph for any given application or library. We augmented Egypt to iterate over every call instruction in the RTL IR and record any native x86-64 `syscall` instruction. Similarly, assembly files also contain `syscall` instruction. Therefore, we analyze the assembly files and extract all `syscall` instructions.

System Call Wrapper Macros In addition to directly using the `syscall` instruction, glibc also uses macro expansion to generate wrappers to system calls. Other glibc routines use these wrappers to invoke system calls. Because these wrappers are implemented as architecture-dependent (in our case x86-64) macros, they cannot be retrieved by analyzing the RTL IR. Moreover, the parameters to these macros are provided by a bash script during compile time.

The `syscall-template.S` file contains the macros `T_PSEUDO`, `T_PSEUDO_NOERRNO`, and `T_PSEUDO_ERRVAL`, that define wrappers to system calls. The list of system calls to be generated, along with other information, such as symbol names and the number of arguments, are provided in the `syscalls.list` file. The Bash script `make-syscalls.sh` reads this file at compile time, generates the correct macro definitions, and invokes the expansion of the macros in the `syscall-template.S`. This script is invoked as part of the build process of glibc. During the compilation of glibc, we trace the execution of this script and record the relevant macro definitions observed during its execution. Using these macros and macro definitions, we derive the mappings between these wrappers and their respective system calls.

Weak Symbols and Versioned Symbols Similarly to musl-libc, glibc uses the `weak_alias` macro to define weak symbols for functions. GCC supports symbol versioning, and glibc uses this feature to support multiple versions of glibc. The versioned symbols are defined using the macro `versioned_symbol`. Both `weak_alias` and `versioned_symbol` provide aliases for functions. Other functions within glibc, as well as the applications using glibc, can invoke these aliased functions either through the original function name or its alias. We analyze the C source code to extract these aliases, and add them to the call graph.

6.2 Binary Analysis

To capture a trace of all invoked executables, we leverage Sysdig [26] to monitor the `execve` calls made during the initial 30 seconds (configurable value) of the container. After we generate the list of programs the container runs, we further perform static analysis to extract the list of system calls necessary for the correct execution of the container.

6.2.1 System Call Invocation Through Libc

After extracting the list of binaries, we recursively find any other libraries (except libc) that are loaded by them, and then use `objdump` to extract the superset of imported functions across all main executables and libraries. This analysis gives us the list of libc (glibc or musl-libc) functions that are imported by an application and its libraries. Then, using the libc-to-syscall map generated as described in Section 6.1, we derive the list of system calls required by the application. In addition to these, the Docker framework itself needs certain system calls to run. Consequently, after deriving the required system calls for all the programs of a container, we combine them with the list of system calls which Docker requires by default to launch the container.

6.2.2 Direct System Call Invocation

We further encountered a limited number of libraries and applications that invoke system calls directly through either the libc `syscall()` interface, or the native `syscall` assembly instruction. Analyzing such invocations requires deriving the values of the arguments being passed to the system call. Fortunately, extracting the first argument, which specifies the system call number, is straightforward, as it is typically set by the (few) instructions preceding the `syscall` instruction or the `syscall()` function invocation. We therefore use binary code disassembly to identify the system call number by extracting the values assigned to the RAX/EAX register for the `syscall` instruction, and the RDI/EDI register for the `syscall()` function. Confine currently supports only x86-64, but adding support for other platforms is straightforward by following their calling conventions.

6.2.3 Dynamically Loaded Libraries

An issue that requires special consideration is dynamic loading, a mechanism through which applications can load modules on demand throughout their execution. The `dlopen()`, `dlsym()`, and `dlclose()` API functions are used to load a library, retrieve its symbols, and close it, respectively. Because these operations are performed at run time, any libraries loaded in this way cannot be identified by looking at the application's ELF binary header. For instance, Apache Httpd uses this feature to load libraries based on the user-defined configuration. Quach and Prakash in [65] have shown that only around 3% of the 3174 programs and 2% of the 4292 libraries analyzed in their dataset used these features, all of which loaded the required libraries during initialization.

To identify such dynamically loaded libraries, we monitor the list of libraries loaded by the application at run time through the `/proc` virtual file system, which provides this information for every process. In Section 6.3 we discuss how we use the same technique of monitoring the `procfs` to detect the list of libraries used by Java applications.

One consideration is that if an application dynamically loads `libc`, we cannot identify the individual functions imported by the application, and would have to retain all system calls made by `libc`. However, it is unlikely that `libc` will be loaded in this fashion, as dynamic loading is used for modules that provide *additional* functionality to the application. We did not encounter any such case in our experiments.

6.3 Language-specific Considerations

Different languages have different software stacks, and therefore different analysis techniques to extract the system call policies. The programming language of the containerized application has an important effect on the analysis methods used to identify the system calls required by a given application. In this section, we describe the different techniques we used to handle applications written in programming languages other than C/C++, which we encountered during our study of the top 200 Docker images. In Section 7 we present statistics on the usage of these languages across the container images studied.

Go Applications written in the Go language consist of *command* packages and utility non-main packages. Go applications can be compiled into executables using two build-modes: *default*, and *c-shared*. When compiled with the *default* build mode, all main packages are built into executables, and all non-main packages are built into a static `.a` archive that is linked statically with the executables. Go applications use system call wrappers provided by Go's `syscall` and `runtime` packages to invoke system calls.

When compiled with the *c-shared* build-mode, the main package relies on the standard `libc` library to invoke system call wrapper functions.

The analysis of our dataset shows that most of the Go applications in the studied containers are built using the default build mode. Therefore, unlike C/C++ applications which rely on `glibc`, these applications use the Go core packages, `syscall` and `runtime`, to make system calls. Consequently, for containers that include Go applications, we require the source code of all running Go applications to identify their system calls. We use the `callgraph` tool [19] to build the call graph of Go applications and all their dependencies, which we have extended to record all calls to the system call wrappers specified in the `syscall` and `runtime` packages.

Java/NodeJS Both the Java and NodeJS runtime applications use `libc` as a shared library to invoke system calls. The Java compiler compiles Java source code into Java bytecode and uses its own virtual machine (JVM) to interpret the bytecode. Java programs are not compiled to machine code and a binary is thus not generated. The interpreter and JVM is provided by the `java` binary, which is launched via an `execve` system call. To find the system calls of a container that hosts a Java type application, in addition to analyzing all other running binaries, we also analyze the `java` binary that contains this JVM, and any other libraries that are dynamically loaded, as described above. Similarly, we handle the system calls invoked by the `node-js` runtime.

Purely Interpreted Languages Scripting languages, such as Python and Perl, are purely interpreted and require the respective interpreter to run. We extract the required system calls for these types of containers using the same approach used for other binaries, by applying our method to the interpreter binary and all its required libraries.

6.4 Seccomp Profile Generation

We automatically generate Seccomp policies by classifying all system calls not present on the final list of required system calls as “not-permitted,” and assigning them to a deny list. The Docker Seccomp ruleset requires the name of the filtered system calls, while our analysis of the containers generates system call numbers. We map all the available system calls in the kernel to their respective number by using the symbol information related to the names of the system calls from the `procfs` pseudo-filesystem. Based on the `sys/syscall.h` header file, we map the system call name to its number, and use it to convert the prohibited system call numbers to their names. We create the Seccomp profile with a deny list containing these system calls and apply it to the container.

Docker uses a JSON file to define the permitted system calls. Listing 1 shows a sample ruleset which only disables the `pwrite64` system call. The default action for this ruleset is to allow all system calls, except those specified under the `syscalls` tag. Each system call is specified by three arguments: its name, the action, and its arguments.

Listing 1: Example of a Docker Seccomp ruleset file.

```
1 {
2   "defaultAction": "SCMP_ACT_ALLOW",
3   "architectures": [
4     "SCMP_ARCH_X86_64",
5     "SCMP_ARCH_X86",
6     "SCMP_ARCH_X32"
7   ],
8   "syscalls": [
9     {
10      "name": "pwrite64",
11      "action": "SCMP_ACT_ERRNO",
12      "args": []
13    }
14  ]
15 }
```

7 Experimental Evaluation

We evaluated Confine with a set of 150 publicly available Docker container images available from Docker Hub [7]. We conducted experiments to get insight into the characteristics of popular container images, assess the effectiveness of our system call policy extraction approach, and evaluate its security benefits in terms of attack surface reduction.

7.1 Analysis of Publicly Available Containers

7.1.1 Dataset Collection

Docker Hub [7] provides a set of 153 “official” container images that are maintained by the service, along with many other community-maintained images. For our evaluation, we collected a data set comprising i) the 153 official Docker Hub images, and ii) the top 200 most popular community-maintained images. Due to the overlap between the two lists, our initial data set consists of 209 unique Docker images.

Out of these 209 images, 193 are available for free and can be used without any paid licensing requirements—we exclude the rest from our data set. Out of the remaining 193 images, 43 require some form of manual effort to setup, such as launching prerequisite containers (e.g., *rocket-chat*), or specifying complex configuration settings. We leave these images out of the scope of our work due to the complexity of the manual steps required to run them. Our final data set thus comprises 150 Docker images (122 official and 28 unofficial) that can be automatically processed by our system.

7.1.2 Container Statistics

Docker Hub assigns a popularity metric to each of the official images based on its number of pull requests (downloads). We retrieve this number through the official Docker Hub

API for the 153 official images (the returned value is zero for most of the non-official images). Figure 3a shows the popularity distribution in terms of number of downloads for the 153 official Docker Hub images. We can see that 15% of the Docker images account for most of the downloads, while the rest are much less popular. This implies that including more images to the evaluation set would not increase the significance of the dataset in terms of container popularity.

In Section 6.3 we discussed how we tackle applications written in languages other than C/C++. To gain some insight about how commonly these languages are used, we gathered some statistics about the programming languages used by the containers in our dataset. The most common programming languages used in containerized applications include C/C++, Java and Go. As shown in Table 1, many well-known server applications (e.g., Nginx, Apache Httpd, and MySQL) fall into the C/C++ category, which comprises 61% of the containers. There are, however, 22% images hosting Java-based applications, such as Apache Cassandra (NoSQL database), Apache Solr (open-source search engine platform), and Apache Tomcat. Since Go has been used to develop many of the management tools used in the Docker ecosystem, Go-based containers also account for a considerable number of images (14%).

We classify containers as *Java-based* or *Python/Perl-based* depending on the presence of the corresponding language runtime in the container. Since most containers contain small utility tools, such as `sed`, `grep`, and `find`, which would cause a container to fall into the C/C++ category, Confine only classifies a container in the C/C++ category if it does not run any other program written in some other programming language. For example, the Cassandra [2] container invokes both `sed` (a C program) and the `java` application. Instead of classifying this container in both the C/C++ and Java categories, we only consider it as a Java container for the results of Table 1.

As discussed in Section 6.2, we extract the list of executed binaries in each container and retrieve their loaded libraries. Figures 3b and 3c show the distribution of the number of binaries executed and libraries loaded by the tested containers. About 75% of the containers execute fewer than 16 binaries, and 75% require fewer than 47 libraries.

7.2 System Call Filtering

We used the set of 150 Docker images to evaluate the effectiveness of our approach in filtering unused system calls. First, Confine automatically analyzes each container and extracts the list of system calls required by its binaries based on the analysis described in Section 6.2. Then, it generates a Seccomp filter to prohibit the use of all remaining system calls. Finally, we run the container on a Docker Engine, along with our filter, to validate the correctness of our analysis.

We assess the effectiveness of our approach by measuring the number of filtered system calls per container. Each system call is an entry point to some kernel functionality,

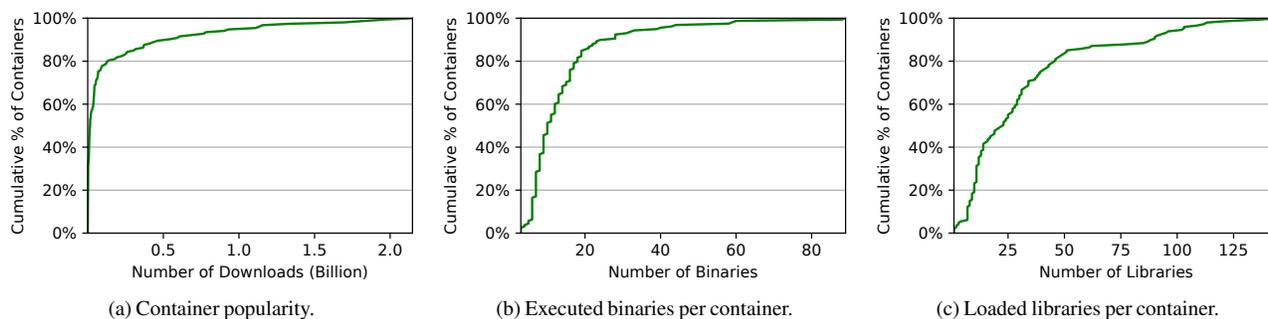


Figure 3: Cumulative distribution of number of downloads, executed binaries, and loaded libraries per container.

Table 1: Breakdown of Docker images according to the programming language used for the main hosted application.

Language	# Img.	Example Applications
C/C++	92	Nginx, Apache Httpd, MongoDB, MySQL
Java	34	Cassandra, Solr, Tomcat, Sonarqube, JRuby
Go	21	Traefik, Registry, Telegraph, Metricbeat
JavaScript	6	Kibana, Ghost, Hipache
Python	5	Celery, Plone, Hylang, Hipache, ROS
Perl	2	Nuxeo, GitLab Community Edition

and thus disabling a system call is equivalent to preventing the exposure of vulnerabilities in all relevant code of that kernel functionality (in addition to prohibiting the use of that system call as part of malicious code)—we have measured the degree of attack surface reduction in terms of known CVEs that become neutralized and present our results in Section 7.3. We leave the actual *removal* of the kernel code related to each system call as part of our future work, but the number of filtered system calls is indicative of the amount of kernel code that could potentially be removed.

Figure 4 shows the cumulative distribution of the number of filtered system calls for the 150 successfully analyzed containers. For about half of the containers, Confine disables 145 or more system calls (out of the 326 currently available in Linux). Even in the worst case (leftmost part of x axis), 100 or more system calls are removed. This means that *at least twice or more* system calls can be disabled compared to Docker’s default Seccomp filter, which includes 49 system calls.

Confine filtered 148 system calls on average for the top-15 ranked Docker images that were successfully analyzed. The list includes many popular applications, such as Nginx, PostgreSQL, MySQL, and MongoDB. For Nginx, we observe that 160 system calls are required for its operation (166 are filtered out of 326), whereas Wan et. al [75] identified only 76 through dynamic analysis. The complete list is shown in Table 2.

7.2.1 Validation Methodology

To ensure that the generated system call policies do not break any functionality, we performed additional validation runs.

Table 2: Number of filtered system calls for the top-15 images with the highest number of downloads.

Image Name	Filtered Syscalls	Popularity Rank
oracle-database-enterprise-edition	138	1
oracle-serverjre-8	190	2
mysql-enterprise-server	128	3
couchbase	140	5
db2-developer-c-edition	100	6
oracle-instant-client	190	7
redis	171	8
ibm-security-access-manager	110	9
mongo	143	10
ubuntu	184	11
busybox	142	12
node	150	13
postgres	133	14
nginx	166	15
mysql	135	16

General Validation First, we check if the container is launched properly with the specified Seccomp profile. Unless we filter system calls that are required by the Docker framework itself, this step will succeed.

The Docker image is specified using a configuration file, called *Dockerfile*. The *Entrypoint* attribute in the Dockerfile specifies the application the container must invoke upon launch. If this application exits (or crashes), the container exits, and thus we verify that this does not happen.

Even if the application remains running, however, it might still encounter errors. For example, it might encounter exceptions that are gracefully handled by the application, but still cause problems in its correct operation. To capture these cases, we check the log files generated by the container. Docker provides a streamlined process of reading the logs produced by the containerized application. We compare the logs produced by the hardened container with the default container. Because values in the logs, such as timestamps and process IDs, might differ between different executions, we ignore these values.

In-Depth Validation We further validated the soundness of the profiles generated for some of the *ready-to-use* Docker images, by testing them with available benchmark suites for the corresponding applications. For this validation effort, we focused on the most popular Docker images due to the manual effort required. We collected a set of benchmarking tools applicable to 10 of the top-50 Docker images. These include domain-specific benchmarking tools, Selenium [24] scripts, and the CloudSuite benchmarks [60]. We applied the benchmarks to the following Docker images: MongoDB, PostgreSQL, MySQL, Redis, Wordpress, PHP, Memcached, Nginx, Apache Httpd and MediaWiki.

The web-serving benchmark of CloudSuite [60] is based on the open-source social networking site Elgg [10]. Elgg is a PHP-based application that uses MySQL as the database server. It also provides a media-streaming server running on the Nginx server. Finally, it also provides a benchmark for Memcached. We derived system call policies for containers running these benchmarks and verified that the benchmarks successfully ran.

For MongoDB, we used `mongo-perf` [15], and applied all the test cases in the `simple_insert` and `simple_query` test suites on one thread for 10 seconds. For PostgreSQL, we used `pgbench` [20], which first creates a new database and then prepares and runs the test cases. For MySQL, we used the `sysbench` tool [48]. After initializing a new database, we ran the OLTP read and write tests using 16 threads on 10 tables. For Redis, we used `redis-benchmark` [22], and ran multiple tests changing the type and number of requests, number of clients and size of data being loaded in each test.

Wordpress and Mediawiki run on Apache Httpd. We ran a hardened MySQL container as their database and used Selenium [24] to create content through automated user interaction. We applied scripts created based on online tutorials prepared and released by Azad et al. [28]. Through these scripts, different operations, such as creating users, posts, applying images and modifying them were performed. All operations were performed successfully and no irregular impact was identified through the logs or the script outputs.

7.2.2 Validation Results

Based on the above process, we verified that 146 out of the 150 containers run successfully, while 4 failed for the following reasons. First, Confine failed to extract system calls for three Go-based images, because the Go call graph tool (discussed in Section 6.3) could not be applied on the source code of Influxdb [12] and Chronograf [4] due to numerous dependencies, while the source code of Sematext [25] was not available.

Second, Java provides the option of accessing OS interfaces directly through its Java Native Interface. We would need to analyze the Java code of each program to extract JNI-invoked system calls. Elasticsearch [9] is the only example we saw in our Java dataset which used this feature, and thus we did not invest the time to implement this capability.

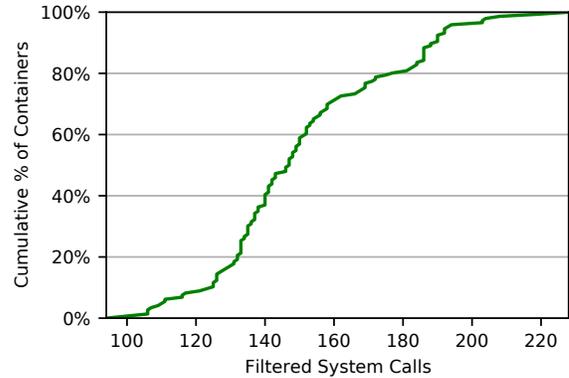


Figure 4: Cumulative distribution of the number of filtered system calls, as a percentage of all tested containers.

7.3 Security Evaluation

Previous works in the area of software debloating [41, 47, 66] mostly focus on the amount of code (and number of ROP gadgets) removed as the main measure of improvement. In contrast, our approach does not remove any code, but merely restricts the system calls that a malicious container can invoke, i.e., it reduces the attack surface of the host’s kernel by reducing the number of system calls it exposes to (potentially malicious) applications.

To demonstrate the effectiveness of our approach in reducing the attack surface, we evaluate how removing unused system calls can reduce the risk of privilege escalation attacks. To that end, our starting point was the vulnerabilities used in a previous study by Lin et al. [54]. These vulnerabilities are exploitable despite the use of container isolation mechanisms, such as namespaces [18], cgroups [3], and capabilities [1]. To gain a better understanding of the impact that filtering individual system calls has in neutralizing potential kernel vulnerabilities, we mapped each CVE to its corresponding system calls.

7.3.1 Mapping Kernel CVEs to System Calls

To perform our analysis, we crawled the CVE website [5] for Linux kernel vulnerabilities using a custom automated tool. The tool parses each commit in the Linux kernel’s Git repository to find the corresponding patch for a given CVE, and retrieves the relevant file and function that was modified by the patch. After mapping CVEs to their respective functions, we built the Linux kernel call graph and analyzed which parts of it can be exclusively accessed by a given system call.

We constructed the Linux kernel’s call graph using KIRIN [78]. This allows us to map which functions in the kernel are invoked from which system call, and therefore reason about which part of the kernel’s code will never be invoked when a set of system calls are filtered.

We discovered that while there are only a few CVEs directly associated with the code of filtered system calls, many CVEs are associated with files and functions that are invoked

Table 3: CVEs mitigated by removing unneeded system calls.

System Call(s)	# CVEs	CVE Examples	CVE Type	# Imgs.	Docker Image Examples
set_thread_area	1	CVE-2014-8133	B	146	Nginx, MongoDB, Apache Httpd, MySQL
mq_notify	1	CVE-2017-11176	D	146	Redis, CouchDB, Apache Httpd, MySQL
sched_getattr	1	CVE-2014-9903	I	146	Postgresql, Nginx, Memcache
io_submit	1	CVE-2010-3066	D	142	Rethinkdb, Apache Httpd, Nginx, Redis
rt_(tg)sigqueueinfo	1	CVE-2011-1182	Other	140	MongoDB, Nginx, Apache Httpd, MySQL
clock_nanosleep	1	CVE-2018-13053	O	140	MongoDB, Nginx, Apache Httpd, MySQL
ioprio_get	1	CVE-2016-7911	P,D	131	Redis, Nginx, Apache Httpd, MySQL
waitid	2	CVE-2017-14954, CVE-2017-5123	B,P,I	114	Nginx, MongoDB, CouchDB, MySQL
inotify_init1	1	CVE-2010-4250	D	101	Nginx, Apache Httpd, MySQL
semctl	1	CVE-2010-4083	I	97	Nginx, CouchDB, Redis
inotify_add_watch	1	CVE-2019-9857	D	77	Nginx, Apache Httpd, MySQL
shmctl	2	CVE-2009-0859, CVE-2010-4072	I,D	71	Iojs, HyLang, Rethinkdb
semget, msgget, shmget	1	CVE-2015-7613	P	62	Julia, Iojs, Clearlinux
splice	1	CVE-2009-1961	D	57	MongoDB, Rethinkdb, Oraclelinux
epoll_ctl	1	CVE-2012-3375	D	48	Crux, IBM-db2-warehouse-cc, Adminer
setsockopt	5	CVE-2016-4997, CVE-2016-8655	P,M,O,D	22	Euleros, Clearlinux
([f,l]remove,[f,l]set)xattr	1	CVE-2011-1090	D	22	Clearlinux, Fluentd
ioctl	26	CVE-2010-2478, CVE-2009-0745	I,P,B,O,D	1	Nats
madvise	2	CVE-2012-3511, CVE-2017-18208	D	1	Busybox

I: Obtain Information, P: Gain Privileges, B: Bypass a Restriction, O: Overflow, D: Denial of Service, M: Memory Corruption

exclusively by the code of filtered system calls. By matching the CVEs to the call graph created by KIRIN, we were able to pinpoint all the vulnerabilities that are related to the set of system calls filtered by a given container. This provides us with a quantifiable property to assess the attack surface reduction achieved by our method, i.e., the number of CVEs that would have been neutralized for a given container, if the respective system call policy generated by Confine was applied.

7.3.2 CVEs Mitigated by Confine

Our results are summarized in Table 3. Linux kernel CVEs are assigned a category depending on how their exploitation can affect the underlying system. While privilege escalation has the most severe outcome, others are important to consider as well. Using a denial-of-service attack, the attacker can disrupt the functionality of *all* containers and applications running on the same host. The “Bypass a Restriction” category includes attacks which allow the attacker to directly or indirectly bypass isolation mechanisms. Similarly, exploiting a vulnerability in the “Obtain Information” category could cause the leakage of sensitive kernel data which endangers the isolation guarantees provided to other containers.

Based on our analysis, in addition to the 25 CVEs mitigated by Docker’s default Seccomp policy, 51 CVEs across all studied containers are effectively removed (i.e., the respective vulnerabilities cannot be triggered by the attacker) by applying our generated policies. These include CVEs that an attacker could exploit to perform denial-of-service attacks against the kernel (CVE-2012-3375, CVE-2016-7911, and CVE-2017-11176), perform privilege escalation attacks (CVE-2017-5123, CVE-2016-7911, and CVE-2015-7613), or leak sensitive kernel information (CVE-2017-14954 and CVE-2014-9903). Of these 51 CVEs, seven were removed in more than 130 containers.

8 Discussion and Limitations

As shown in Table 3, the system calls filtered by our technique are not very commonly used, but at the same time mitigate a large number of previously disclosed kernel vulnerabilities. We must emphasize that although system calls such as `execve` and `mmap` are used as part of user-space exploits, *any* system call associated with a kernel CVE can be used to exploit the kernel. For an attacker seeking to escape a container, exploiting commonly used system calls such as `execve` or `mmap` provides no additional benefit over exploiting system calls such as `waitid`, which are used less frequently.

In addition to launching applications from scripts and the command line, most programming languages give the programmer the ability to launch applications using special library calls, such as `execve`. As it is not guaranteed that such invocations will occur within our monitoring window, our approach may fail to analyze any executables launched in this way. Currently, the developer is expected to provide a list of binaries executed using such library calls. Our approach provides an initial list of applications for the user to build upon, which can further reduce the manual effort required.

A better alternative would be to statically analyze the source code of all invoked applications to identify process creation events. This can easily be done for applications written in interpreted languages, as they are typically supported by many static analysis tools (e.g., `php-ast` [62] for PHP, or the built-in AST [21] functionality for Python). We executed `php-ast` on the Wordpress Docker image and validated the correctness of extracting paths of binaries which could be passed to any `exec`-like function (e.g., `php_exec`, `shell_exec`). This could easily be extended to applications written in different languages. We leave the full implementation of such a capability as part of our future work, since it only requires engineering effort.

Although not recommended, some Docker images use `cron` jobs to run periodic tasks in the container. In these cases we expect the user to provide the list of programs which can be executed through `cron`, although again such cases could be automatically handled by parsing the `crontab` file.

9 Related Work

Static source code analysis for deriving system call policies has been a widely used approach in the fields of sandboxing and host-based intrusion detection [33–35, 43, 49, 61, 67, 74]. Our work mainly falls in the area of software debloating, and we thus discuss related works in this context.

9.1 Container Security and Debloating

Wan et al. [75] use dynamic analysis to profile the running applications on a container and generate corresponding Seccomp filters. DockerSlim [8] is an open source tool which also relies on dynamic analysis to generate Seccomp profiles and to remove unnecessary files from docker images. As discussed in Section 7.2.2, all required system calls cannot be reliably extracted through dynamic analysis alone—especially for cases that handle exceptions and errors, which are typically not part of the common execution paths. Therefore, dynamic analysis cannot guarantee complete coverage of all the system calls required by each application. Our system, on the other hand, provides a more comprehensive static analysis mechanism for extracting the system calls used by a container.

Speaker [52] separates the required system calls into two main phases, booting and runtime. It dynamically extracts the required system calls for each phase and filters them based on the necessity of each state. Cimplifier [68] splits containers running multiple applications into multiple single-purpose containers using dynamic analysis. Rastogi et al. [69] propose improvements to Cimplifier [68] through symbolic execution.

Previous works have also focused on container security from the perspective of software protection mechanisms and vulnerabilities. Lin et al. [54] provide a dataset of security vulnerabilities and exploits which can potentially bypass the software isolation provided by the Linux Kernel. One of their recommendations is the use of stricter Seccomp policies for containers. Shu et al. [71] have created a framework for performing vulnerability scanning on images found on Docker Hub. Combe et al. [31] explored the security implications of using containers, by considering adversary models which assume complete access of an adversary to one container on a host.

9.2 Application Debloating

Most of the prior works in the area of debloating have focused on removing unnecessary code from individual processes. Mulliner and Neugschwandtner [58] proposed one of the first approaches for library specialization, which identifies

and removes all non-imported library functions at load time. Quach et al. [66] developed a modified loader and compiler to perform shared library specialization by removing unnecessary functions extracted through call dependency and function boundary identification at compile time. Agadakos et al. [27] perform similar library specialization, but at the binary level. BlankIt [63] only loads library functions upon request, keeping the program and only parts of the library which are required at that moment in the process address space. Song et al. [72] used data dependency analysis to show the potential of fine-grained library customization of statically linked libraries. Shredder [56] and Saffire [57] restrict the arguments passed to critical system API functions to only legitimate values extracted from each application’s code. Qian et al. [64] use training and heuristics to identify basic blocks which can be removed from a binary, while Ghaffarinia and Hamlen [36] restrict the control flow of the binary, instead of removing the extra code using a similar training approach.

Sysfilter [32] uses binary analysis to identify the set of required system calls for a given application, and restricts access to them through binary rewriting performed by Egalito [76]. While Confine mainly relies on the libc call graph generated through source code analysis, Sysfilter relies on the binary, which could cause loss of precision and overapproximation in identifying the set of required system calls. Temporal system call specialization [37] disables system calls according to the execution phase of server applications, allowing many security-critical system calls (e.g., `execve`) to be disabled after the application finishes its initialization phase.

Other works explore the potential of software debloating based on predefined features. CHISEL [41] is a framework for shrinking software using a reinforcement learning approach based on test cases provided by the user. The overall approach is driven by the test cases, reducing the code size while ensuring that none of the test cases fail. TRIMMER [39] uses inter-procedural analysis to find unnecessary parts of code based on user-defined configuration data.

Other works in this area have also focused on different programming languages [44, 73, 77]. Jred [77] performs static analysis on Java code to remove unused methods and classes. Jiang et al. [44] propose feature-based debloating for Java programs using data flow analysis.

9.3 Kernel Debloating

Several works focus on debloating the kernel and customizing it according to user requirements. KASR [79] and FACE-CHANGE [80] use dynamic analysis to identify unused parts of the kernel and use virtualization mechanisms to limit each application to its profile. Kurmus et al. [50] propose a method for tailoring the Linux kernel to special workloads through automatic generation of kernel configuration files.

10 Conclusion

Our work was motivated by the lack of a generic solution for the automated generation of restrictive system call policies for container environments—one that does not rely on training with realistic workloads, which is a cumbersome and error-prone method. We believe that the results of our experimental evaluation demonstrate the practicality of the proposed approach, as Confine managed to disable (without breaking any functionality) 145 or more system calls for more than half of the analyzed containers, neutralizing this way 51 previously disclosed kernel vulnerabilities. As part of our future work, we plan to address the limitations of our prototype and explore the generation of more fine-grained system call policies.

Acknowledgments

This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045, with additional support by Accenture. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, DARPA, or Accenture.

References

- [1] Capabilities(7) - Linux Programmer's Manual. <http://man7.org/linux/man-pages/man7/capabilities.7.html>.
- [2] Cassandra - Docker Hub. https://hub.docker.com/_/cassandra/.
- [3] Cgroups(7) - Linux Programmer's Manual. <http://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [4] Chronograf - Docker Hub. https://hub.docker.com/_/chronograf.
- [5] Common vulnerabilities and exposures database. <https://www.cvedetails.com>.
- [6] CVE-2017-5123. <https://www.cvedetails.com/cve/CVE-2017-5123/>.
- [7] Docker Hub. <https://hub.docker.com>.
- [8] DockerSlim. <https://dockersl.im>.
- [9] Elasticsearch - Docker Hub. https://hub.docker.com/_/elasticsearch.
- [10] Elgg. <https://elgg.org/>.
- [11] GNU Compiler Collection. <https://gcc.gnu.org>.
- [12] Influxdb - Docker Hub. https://hub.docker.com/_/influxdb.
- [13] Kubernetes - Production-Grade Container Orchestration. <https://kubernetes.io>.
- [14] The LLVM compiler infrastructure. <http://llvm.org>.
- [15] Mongo-perf. <https://github.com/mongodb/mongo-perf>.
- [16] MongoDB - Docker Hub. https://hub.docker.com/_/mongo/.
- [17] Musl Libc. <https://www.musl-libc.org>.
- [18] Namespaces(7) - Linux Programmer's Manual. <http://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [19] Package Callgraph - GoDoc. <https://godoc.org/golang.org/x/tools/go/callgraph>.
- [20] Pgbench. <https://www.postgresql.org/docs/10/pgbench.html>.
- [21] Python AST. <https://docs.python.org/3/library/ast.html>.
- [22] Redis-benchmark. <https://redis.io/topics/benchmarks>.
- [23] Seccomp BPF (SECure COMPUting with filters). https://www.kernel.org/doc/html/v4.16/user-space-api/seccomp_filter.html.
- [24] Selenium. <https://selenium.dev/>.
- [25] Sematext Agent Monitoring and Logging - Docker Hub. https://hub.docker.com/_/sematext-agent-monitoring-and-logging.
- [26] Sysdig. <https://github.com/draios/sysdig>.
- [27] Ioannis Agadacos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.
- [28] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [29] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

- [30] Brandon Butler. Which is cheaper: Containers or virtual machines? <https://www.networkworld.com/article/3126069/which-is-cheaper-containers-or-virtual-machines.html>, September 2016.
- [31] Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.
- [32] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.
- [33] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 194–208, 2004.
- [34] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 120–128, 1996.
- [35] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.
- [36] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.
- [37] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.
- [38] Andreas Gustafsson. Egypt. <https://www.gson.org/egypt/egypt.html>.
- [39] Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.
- [40] Haifeng He, Saumya K Debray, and Gregory R Andrews. The revenge of the overlay: automatic compaction of OS kernel code via on-demand code loading. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 75–83, 2007.
- [41] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.
- [42] Michael Hind. Pointer analysis: Haven’t we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.
- [43] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.
- [44] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.
- [45] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.
- [46] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, pages 957–972, 2014.
- [47] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.
- [48] Alexey Kopytov. Sysbench. <https://github.com/akopytov/sysbench>.
- [49] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, 2005.
- [50] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.
- [51] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An application-oriented Linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 20(6):1093–1107, 2004.
- [52] Linguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-phase execution of application containers. In

Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pages 230–251, 2017.

- [53] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.
- [54] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 418–429, 2018.
- [55] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference*, 1993.
- [56] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.
- [57] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.
- [58] Collin Mulliner and Matthias Neugschwandtner. Breaking payloads with runtime code stripping and image freezing, 2015. Black Hat USA.
- [59] Karen Scarfone Murugiah Souppaya, John Morello. Application Container Security Guide, 2017. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf>.
- [60] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, 2016.
- [61] Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 156–167, 2008.
- [62] Nikita Popov. PHP abstract syntax tree. <https://github.com/nikic/php-ast>.
- [63] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–180, 2020.
- [64] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, 2019.
- [65] Anh Quach and Aravind Prakash. Bloat factors and binary specialization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 31–38, 2019.
- [66] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.
- [67] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 358–367, 2005.
- [68] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.
- [69] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2nd Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 51–56, 2017.
- [70] Daniel Shapira. Escaping Docker container using waitid() – CVE-2017-5123, 2017. <https://www.twistlock.com/labs-blog/escaping-docker-container-using-waitid-cve-2017-5123/>.
- [71] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on Docker Hub. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 269–280, 2017.
- [72] Linhai Song and Xinyu Xing. Fine-grained library customization. In *Proceedings of the 1st ECOOP International Workshop on Software Debloating and Layering (SALAD)*, 2018.
- [73] Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.
- [74] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 156–168, 2001.

- [75] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.
- [76] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, 2020.
- [77] Dinghao Wu Yufei Jiang and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.
- [78] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium*, pages 1205–1220, 2019.
- [79] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 691–710, 2018.
- [80] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

sysfilter: Automated System Call Filtering for Commodity Software

Nicholas DeMarinis Kent Williams-King Di Jin
Rodrigo Fonseca Vasileios P. Kemerlis

*Department of Computer Science
Brown University*

Abstract

Modern OSes provide a rich set of services to applications, primarily accessible via the system call API, to support the ever growing functionality of contemporary software. However, despite the fact that applications require access to part of the system call API (to function properly), OS kernels allow full and unrestricted use of the entire system call set. This not only violates the principle of least privilege, but also enables attackers to utilize extra OS services, after seizing control of vulnerable applications, or escalate privileges further via exploiting vulnerabilities in less-stressed kernel interfaces.

To tackle this problem, we present `sysfilter`: a binary analysis-based framework that automatically (1) limits what OS services attackers can (ab)use, by enforcing the principle of least privilege with respect to the system call API, and (2) reduces the attack surface of the kernel, by restricting the system call set available to userland processes. We implement `sysfilter` for x86-64 Linux, and present a set of program analyses for constructing system call sets statically, and in a scalable, precise, and complete (safe over-approximation) manner. In addition, we evaluate our prototype in terms of correctness using 411 binaries (real-world C/C++ applications) and $\approx 38.5\text{K}$ tests to assert their functionality. Furthermore, we measure the impact of our enforcement mechanism(s), demonstrating minimal, or negligible, run-time slowdown. Lastly, we conclude with a large scale study of the system call profile of $\approx 30\text{K}$ C/C++ applications (from Debian `sid`), reporting insights that justify our design and can aid that of future (system call-based) policing mechanisms.

1 Introduction

Software is continuously growing in complexity and size. `/bin/true`, the “tiny” command typically used as aid in shell scripts, was first introduced in the 7th edition of the Unix distribution (Bell Labs) and consisted of *zero* lines of code (LOC); by 2012, in Ubuntu, `true` has grown up to 2.3 KLOC [28]. Likewise, the `bash` binary has gone from 11.3 KB (Unix V5, 1974) up to 2.1 MB (Ubuntu, 2014) [28].

This constant stream of additional functionality integrated into modern applications, i.e., *feature creep*, not only has dire effects in terms of security and protection [1, 71], but also necessitates a rich set of OS services: applications need to interact with the OS kernel—and, primarily, they do so via the system call (syscall) API [52]—in order to perform useful tasks, such as acquiring or releasing memory, spawning and terminating additional processes and execution threads, communicating with other programs on the same or remote hosts, interacting with the filesystem, and performing I/O and process introspection.

Indicatively, at the time of writing, the Linux kernel (v5.5) provides support for 347 syscalls in x86-64. However, not every application requires access to the complete syscall set; e.g., $\approx 45\%/65\%$ of all (C/C++) applications in Debian “`sid`” (development distribution) [86] do not make use of `execve/listen` in x86-64 Linux. In other words, roughly one-half of these applications do not require the ability (and should not be permitted) to invoke other programs or accept network connections. Alas, the OS kernel provides *full* and *unrestricted* access to the entirety set of syscalls. This is not only a violation of the *principle of least privilege* [76], but also allows attackers to: (a) utilize additional OS services after seizing control of vulnerable applications [69], and (b) escalate their privileges further via exploiting vulnerabilities in unused, or less-stressed, OS kernel interfaces [33–35, 43, 65, 66].

To mitigate the effects of (a) and (b) above, we present `sysfilter`: a framework to (automatically) limit what OS services attackers can (ab)use, by enforcing the principle of least privilege with respect to the syscall API [69], and reduce the *attack surface* of the OS kernel, by restricting the syscall set available to userland processes [43]. `sysfilter` consists of two parts: system call set extraction and system call set enforcement. The former receives as input a target application, in binary form, automatically resolves dependencies to dynamic shared libraries, constructs a *safe*—but *tight*—over-approximation of the program’s function-call graph (FCG), and performs a set of program analyses atop the FCG, in order to extract the set of developer-intended syscalls.

The latter enforces the extracted set of syscalls, effectively sandboxing the input binary. We implemented `sysfilter` in x86-64 Linux, atop the Egalito framework [95], while our program analyses, crafted as “passes” over Egalito’s intermediate representation, are *scalable*, *precise*, and *complete*. `sysfilter` can extract a tight over-approximation of the set of developer-intended syscalls for $\approx 90\%$ of all C/C++ applications in Debian `sid`, in less than 200s (or for $\approx 50\%$ of these apps in less than 30s; § 5). Moreover, `sysfilter` requires no source code (i.e., it operates on stripped binaries, compiled using modern toolchains [26, 67]), and can sandbox programs that consist of components written in different languages (e.g., C, C++) or compiled-by different frameworks (GCC, LLVM). Importantly, `sysfilter` does not rely (on any form of) dynamic testing, as the results of this approach are usually both unsound and incomplete [62].

Further, we evaluate `sysfilter` across three dimensions: (1) correctness, (2) performance overhead, and (3) effectiveness. As far as (1) is concerned, we used 411 binaries from various packages/projects, including the GNU Coreutils (100, 672), SPEC CINT2006 (12, 12), SQLite (7, 31190), Redis (6, 81), Vim (3, 255), Nginx (1, 356), GNU M4 (1, 236), GNU Wget (1, 130), MariaDB (156, 2059), and FFmpeg (124, 3756), to extract and enforce their corresponding syscall sets; once sandboxed, we stress-tested them with $\approx 38.5\text{K}$ tests. In all cases, `sysfilter` managed to extract a complete and tight over-approximation of the respective syscall sets, demonstrating that our prototype can successfully handle complex, real-world software. (The numbers *A*, *B* in parentheses denote the number of binaries analyzed/enforced and the number of tests used to stress-test them, respectively.)

Regarding (2), we used SPEC CINT2006, Nginx, and Redis—i.e., 19 program binaries in total. In all cases, the sandboxed versions exhibited minimal, or negligible, run-time slowdown due to syscall filtering; we explored a plethora of different settings and configurations, including interpreted vs. JIT-compiled filters, and filter code that implements sandboxing using a linear search (within the respective syscall set) vs. filter code that utilizes a skip list-based approach. Lastly, with respect to (3), we investigated how `sysfilter` can reduce the attack surface of the OS, by inquiring what percentage of all C/C++ applications in Debian `sid` ($\approx 30\text{K}$ binaries in total) can exploit 23 Linux kernel vulnerabilities after hardened with `sysfilter`. Although `sysfilter` does not defend against control- or data-flow hijacking [87] our results demonstrate that it can mitigate attacks by means of least privilege enforcement and (OS) attack surface reduction.

We conclude our work with a large scale study of the syscall sets of $\approx 30\text{K}$ C/C++ applications (Debian `sid`), reporting insights regarding the syscall set sizes (i.e., the number of syscalls per binary), most- and least-frequently used syscalls, syscall site distribution (libraries vs. main binary), and more. The results of this analysis not only guide our design, but can also aid that of future syscall policing mechanisms.

2 Background and Threat Model

Adversarial Capabilities In this work, we consider userland applications that are written in memory-unsafe languages, such as C/C++ and assembly (ASM). The attacker can trigger vulnerabilities, either in the main binaries of the applications or in the various libraries the latter are using, resulting in memory corruption [87]. Note that we do not restrict ourselves to specific kinds of vulnerabilities (e.g., stack- or heap-based memory errors, or, more generally, spatial or temporal memory safety bugs) [59, 60] or exploitation techniques (e.g., code injection, code reuse) [13, 78, 79, 87, 97].

More specifically, the attacker can: (a) trigger memory safety-related vulnerabilities in the target application, multiple times if needed, and construct and utilize exploitation primitives, such as arbitrary memory writes [16] and reads [81]; and (b) use, or combine, such primitives to tamper-with critical data (e.g., function and `vtable` pointers, return addresses) for hijacking the control flow of the target application and achieve arbitrary code execution [83] via means of code injection [97] or code reuse [10, 13, 24, 29, 32, 78, 79]. In terms of adversarial capabilities, our threat model is on par with the state of the art in C/C++/ASM exploitation [41, 87]. Lastly, we assume that the target applications consist of *benign* code: i.e., they do not contain malicious components.

Hardening Assumptions The primary focus of this work is modern, x86-64 Linux applications, written in C, C++, or ASM (or any combination thereof), and compiled in a *position-independent* [64] manner via toolchains that (by default) *do not mix code and data* [3, 4], such as GCC and LLVM.¹ In addition to the above, we assume the presence of *stack unwinding information* (`.eh_frame` section) in the ELF [12] files that constitute the target applications.

In § 3, we explain in detail the reasons for these two requirements—i.e., position-independent code (PIC) and `.eh_frame` sections. However, note that (a) PIC is enabled by default in modern Linux distributions [14, 95], while (b) `.eh_frame` sections are present in modern GCC- and LLVM-compiled code [3, 95]. The main reason for (a) is full ASLR (Address Space Layout Randomization): in position-dependent executables ASLR will only randomize the process stack and `mmap`- and `brk`-based heap [7]. Moreover, PIC in x86-64 incurs negligible performance overhead due to the existence of PC-relative data transfer instructions (`%rip`-relative `mov`) and extra general-purpose registers (16 vs. 8 in x86). As far as (b) is concerned, stack unwinding information is mandated by C++ code for exception handling [49], while both GCC and LLVM emit `.eh_frame` sections even for C code to support interoperability with C++ [95] and various features of certain `libc` (C library) implementations—e.g., `backtrace` in `glibc` (GNU C Library).

¹Andriess et al. [4], and Alves-Foss and Song [3], have independently verified that modern versions of both GCC and LLVM *do not mix code and data*. `icc` (Intel C++ Compiler) still embeds data in code [3].

Lastly, we assume a Linux kernel with support for `seccomp-BPF` (SECure COMPUting with filters) [36]. (All versions \geq v3.5 provide support for `seccomp-BPF` in x86-64.) Every other standard userland hardening feature (e.g., NX, ASLR, stack-smashing protection) is orthogonal to `sysfilter`; our proposed scheme does not require nor preclude any such feature. The same is also true for less-widespread mitigations, like CFI [9,96], CPI [40,53], code randomization/diversification [38,41,94], and protection against data-only attacks [68].

3 Design and Implementation

Approach `sysfilter` aims at mitigating the effects of application compromise by *restricting access to the syscall API* [52]. The benefits of this approach are twofold: (1) it limits post-exploitation capabilities [69], and (2) it prevents compromised applications from escalating their privileges further via exploiting vulnerabilities in unused, or less-stressed, kernel interfaces [33–35,43,65,66].

The main idea behind (1) is that applications need to interact with the kernel—and they primarily do so via the syscall API—in order to perform useful tasks. Indicatively, at the time of writing, the Linux kernel (v5.5) provides support for 347 syscalls in x86-64. (This number does *not* include the syscalls needed for executing 32-bit x86 applications atop a 64-bit kernel, or 64-bit processes that adhere to the x32 ABI [50].) However, despite the fact that applications only require access to part of the aforementioned API to function properly (e.g., non-networked applications do not need access to the `socket`-related syscalls), the OS kernel provides full and unrestricted access to the entirety set of syscalls.

This approach violates the *principle of least privilege* [76] and enables attackers to utilize additional OS services after seizing control of vulnerable applications. By restricting access to certain syscalls, `sysfilter` naturally limits what OS services attackers can (ab)use and enforces the principle of least privilege with respect to the syscall API: i.e., programs are allowed to issue only developer-intended syscalls.

As far as (2) is concerned, multiple studies have repeatedly divulged that the exploitation of vulnerabilities in kernel (or in even lower-level, more-privileged [19,99]) code is an essential part of privilege escalation attacks [33–35,43,65,66]. To this end, `sysfilter` reduces the *attack surface* of the OS kernel, by restricting the syscall set available to userland processes, effectively providing *defense-in-depth* protection.

Overview `sysfilter` consists of two parts (see Figure 1): (1) a syscall-set *extraction* component; and (2) a syscall-set *enforcement* component. The former receives as input the target application in binary form (ELF file), automatically resolves all dependencies to dynamic shared libraries (`.so` ELF objects), and constructs a *safe* over-approximation of the program’s FCG—across all objects in scope. Finally, it performs a set of program analyses atop FCG, in order to make

the over-approximation as tight as possible and construct the syscall set in question. Note that the tasks above are performed *statically* and the syscall set returned by the extraction tool is *complete*: i.e., under any given input, the syscalls performed by the corresponding process are guaranteed to exist in the syscall set—this includes syscalls that originate from the binary itself, `libc`, or any other dynamically-loaded shared library. The latter part enforces the extracted set of syscalls, effectively *sandboxing* the input binary. Specifically, given a set of syscall numbers, the enforcement tool converts them to a BPF program [54] to be used with `seccomp-BPF` [36].

3.1 System Call Set Extraction

3.1.1 Analysis Scope

The input to the syscall-set extraction component of `sysfilter` is an (x86-64) ELF file that corresponds to the main binary of the application (see Figure 1A). `sysfilter` requires PIC as input, which is the default setting in modern Linux distributions [14,95]. Once `sysfilter` verifies that the main binary is indeed PIC, adds it to the *analysis scope*, and proceeds to resolve dependencies regarding dynamic shared libraries. This is accomplished by first checking the `PT_INTERP` header, which conventionally contains the path of the respective dynamic linker/loader (e.g., `/lib64/ld-linux-x86-64.so`), followed by iterating the `.dynamic` ELF section for `DT_NEEDED` entries that correspond to the names of the required shared libraries. Each of these libraries is added to the analysis scope and their `.dynamic` section is also scanned, recursively, to recover additional (library) dependencies. The process stops when all *explicit* dynamic library dependencies are resolved and the related ELF files have been added to the analysis scope.

In addition to the above, it is also possible to provide as input a set of *implicit* dynamic shared object dependencies to `sysfilter`: i.e., a list of additional `.so` ELF files that need to be added to the analysis scope, irrespectively of whether they exist in any of the loaded objects’ `.dynamic` section (see Figure 1B). This functionality is important in order to support the analysis of binaries that have run-time dependencies to shared objects (e.g., via `dlopen`) or use `LD_PRELOAD`.

3.1.2 Function-Call Graph Construction

Once every ELF object is added to the analysis scope, `sysfilter` proceeds with the construction of the *function-call graph* (FCG) of the *whole* program. The FCG contains parts of the code (functions) that are reachable, under any possible input to the corresponding process. Note that for every included `.so` ELF object in the analysis scope, not all of their code is used: e.g., applications that link with `libc`, `libpthread`, `libdl`, *etc.*, do not make full use of the latter; usually, only part of library functionality is utilized [1,71].

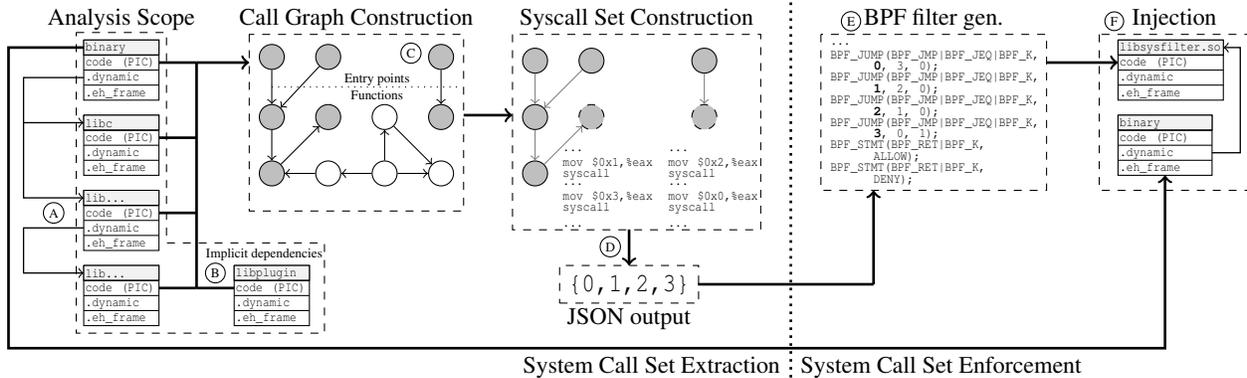


Figure 1: **The `sysfilter` Architecture.** The tool consists of two parts: the system call set extraction part (left) and the system call set enforcement part (right). The former receives as input the target application in binary form, automatically resolves all dependencies to dynamic shared libraries, constructs a safe over-approximation of the program’s FCG—across all objects in scope—and, finally, performs a set of program analyses atop the FCG, in order to extract the set of developer-intended syscalls. The latter enforces the extracted set of syscalls, effectively sandboxing the input binary, using `seccomp`-BPF.

Precise Disassembly Obtaining the complete and precise disassembly of arbitrary binary programs is an undecidable task [91]. The problem stems from two main reasons: (a) not being able to decisively differentiate code from data [91]; and (b) not being able to precisely identify function boundaries [5, 6]. Fortunately, modern toolchains, like GCC and LLVM, (1) do not mix code and data [3, 4] and (2) embed stack unwinding information to (x86-64) C/C++ binaries [95]. `sysfilter` takes advantage of (1) and (2) in order to precisely disassemble the executable code from all ELF files in the analysis scope, *without requiring symbols or debugging information* (If such information is available, `sysfilter` will use it, but our techniques are designed for stripped binaries.)

More specifically, for each `.so` object in the analysis scope, `sysfilter` uses the stack unwinding information (`.eh_frame` section) to get the exact boundaries of all functions in executable sections (e.g., `.text`, `.plt`). Moreover, special care is taken to correctly identify functions of `crtstuff.c` (`libgcc`), which are compiled into the sections `.init` and `.fini`, as well as into `crtbegin.o`. Armed with precise information about function boundaries, and given the strict separation of code and data, `sysfilter` performs a *linear sweep*, in all code regions that correspond to identified functions, to disassemble their executable code. The resulting disassembly does not contain any invalid instruction, due to data treated as code or incorrect function boundary detection, nor it misses instructions due to unidentified code—the resulting disassembly is complete, precise, and accurate.

Direct Call Graph Building Upon the precise disassembly obtained during the previous step, `sysfilter` proceeds to construct the FCG of the input program. First, it puts together the *direct call graph* (DCG): i.e., the part of the FCG that corresponds to directly-invoked functions/code. This is achieved by first adding to the DCG the entry point of the main binary,

followed by all the functions whose addresses are stored at the subsequent (ELF) sections: `.preinit_array`, `.init_array`, and `.fini_array`; the code/function in `.init` and `.fini` is also added to the DCG. Subsequently, the same process is repeated for every other `.so` ELF object in the analysis scope. At the end of this step, a set of *initial* functions are added to the DCG, which correspond to the entry points of the code that is executed during the initialization/finalization of the respective process by the dynamic linker/loader (`ld.so`). It is also possible to provide as input a set of implicit function dependencies (see Figure 1B). Again, this is required to aid the analysis of binaries that have run-time dependencies to certain functions (e.g., via `dlsym`) or use `LD_PRELOAD`.

Next, the code of each such function in the DCG is scanned, linearly, to identify direct call instructions that target other functions in the respective ELF objects. Branch instructions, like (un)conditional `jmp`, which cross function boundaries are also taken into consideration as they are typically used for implementing *tail-call elimination* [84]. Each identified target function (callee) is also added to the DCG, and the process is repeated until no additional functions can be added. Cross-shared library calls, via the Procedure Linkage Table (PLT), are handled by inspecting the `.dynsym`, and `.dynstr`, sections of the ELF files in scope and “emulating” the binding (symbol resolution) rules of `ld.so`.² (Direct cross-`.so` object calls via PLT are treated as direct intra-`.so` function calls.)

The net result of the above is the construction of the part of the FCG that contains the entry point(s) and the initialization/finalization functions of the ELF objects in scope (plus the implicitly-added functions, if provided), followed by every other function that is *directly-reachable* from them (i.e., reachable by following the targets of direct `call/jmp` instructions and resolving PLT entries).

²This is analogous to executing `ld.so` with `'LD_BIND_NOW=1'`.

Address-taken Call Graph The aforementioned process does not take into consideration functions targeted-by indirect `call/jmp` instructions. Such instructions have as operand a (general-purpose) register, or a memory location, which stores the target address (i.e., address of the callee), and are typically used for dereferencing function pointers (C/C++) and implementing dynamic dispatch (C++) [77]. Resolving the target addresses of indirect `call/jmp` instructions, statically, is a hard problem [72], mostly due to the imprecision of *points-to* analysis [44]. (Note that resolving target addresses using dynamic testing is even more problematic, as the results of this approach usually lack *soundness* [62].) Starting with DCG, `sysfilter` proceeds to *over-approximate* the FCG by constructing, what we refer to as, the *address-taken call graph* (ACG). The process of constructing the ACG is *complete*: i.e., it never excludes functions that can be executed by the program (under any possible input).

The first step in the construction of the ACG is the identification of all *address-taken* functions: i.e., functions whose address appears in `rvalue` expressions, function arguments, `struct/union` initializers, and C++ object initializers, or functions that correspond to *virtual* methods (C++). The set of all address-taken (AT) functions is a superset of the possible targets of every indirect call site in scope. This is because indirect `call/jmp` instructions take as operands (general-purpose) registers, or memory locations, which can only hold absolute addresses; therefore, in order for a function to end-up being invoked via an indirect `call/jmp` instruction, its address must first be “taken”, and then loaded in the respective operand (be it a register or memory location).

`sysfilter` leverages the fact that every ELF object in the analysis scope is compiled as PIC, in order to identify all AT functions. More specifically, locations in code, or data, ELF regions that correspond to absolute function addresses must always have accompanying relocation entries (`relocs`), if PIC is enabled [14]. `sysfilter` begins with identifying all the relocation sections (i.e., sections of type `SHT_REL` or `SHT_RELA`) in the ELF objects in scope. Next, it processes all the `relocs`, searching for cases where the computation of the relocation involves the starting address of a function (recall that we have already identified the boundaries of every function in scope, during the construction of the DCG). Every such function, whose starting address is used in `relocs` computation, is effectively an AT function. The same function can have its address taken multiple times in different locations (e.g., function arguments, `rvalue` expressions in function bodies, or as part of global `struct/union/C++` object initializers). Relocations that are applied to special sections (e.g., `.plt`, `.dynamic`) are ignored, as they are only related to dynamic binding.

Armed with the set of all AT functions, `sysfilter` proceeds with computing the reachable functions from (each one of) them using the same approach we employed for constructing the DCG. ACG effectively contains as “entry points” the discovered AT functions, followed by every other function that

is directly-reachable from them. The combined set of functions in DCG and ACG is a superset of the set of functions in the program’s FCG: i.e., $V[FCG] \subseteq (V[DCG] \cup V[ACG])$.

Vacuumed Call Graph Although $DCG \cup ACG$ is a safe over-approximation of FCG, it is not a tight one, as every AT function included in the considered call graph is (potentially) “polluting” it in a considerable manner by bringing in scope every other function that is reachable from itself. In order to keep the over-approximation as tight as possible, `sysfilter` *prunes* the ACG using a technique for software debloating [1, 71]. In particular, we begin with the observation that each time the address of a function is taken, a code pointer is created. By taking into account the location (ELF section) that such code pointers are created, `sysfilter` further separates those found in code (e.g., `.text`) and data (e.g., `.rodata`) regions. For the former, it iterates every function that has been deemed as unreachable, and checks if the address of an AT function is only taken within functions that are (strictly) not part of the call graph. If this condition is true, it removes the respective AT function from ACG, which may result in additional removals (e.g., everything directly-reachable from the removed AT function); `sysfilter` iteratively performs the above until no additional functions can be pruned.

For the latter case, `sysfilter` cannot prune AT functions using the same approach, as the resulting code pointers can be part of encapsulating data structures whose usage cannot be tracked without access to symbol (or debugging) information. However, if such information is indeed available—note that modern toolchains (GCC, LLVM) include symbols in the resulting ELF objects (`.symtab` section) by default, while popular Linux distributions provide symbols for their packaged binaries—, `sysfilter` can (more aggressively) eliminate AT functions from data sections as follows. First, it leverages symbol information to identify the bounds (`[OBJ_BEGIN - OBJ_END]`) of global data objects (i.e., symbols of type `OBJECT/GLOBAL`). Next, it checks for `relocs` that: (a) correspond to AT functions; and (b) fall within the bounds of any global object. The net result of this approach is the identification of statically-initialized arrays of code pointers or data structures that contain code pointers. Lastly, `sysfilter` iterates every function that has been classified as unreachable, and checks if `OBJ_BEGIN` is taken only within such functions. Again, if this condition is true, it marks the AT functions that correspond to the object beginning at `OBJ_BEGIN` as unreachable, and iteratively performs the purging process until no additional functions can be classified as unreachable. We refer to the pruned ACG, combined with DCG, as *vacuumed call graph* (VCG). Specifically, $VCG = DCG \cup ACG'$, where ACG' denotes the pruned ACG using the approach outlined above (see Figure 1C). Again, VCG is a complete, tight over-approximation of the true FCG: i.e., `sysfilter` only excludes functions that can never be executed by the program (under any possible input); more formally: $V[FCG] \subseteq V[VCG] \subseteq (V[DCG] \cup V[ACG])$.

```

1 #define ctor __attribute__((constructor))
2 typedef void (*fptr)(void);
3
4 void f10(void) { ... }
5 ctor void f9(void) { ... f10(); ... }
6 void f8(void) { ... }
7 void f7(void) { ... f8(); ... }
8 void f6(void) { ... }
9 void f5(void) { ... fp_arr[n](); ... }
10 void f4(void) { ... f5(); ... }
11 void f3(void) { ... }
12
13 fptr f2(void) { ... return &f4; }
14 fptr f1(void) { ... return &f3; }
15
16 fptr fp;
17 fptr fp_arr[] = {&f6, &f7};
18
19 int main(void)
20 {
21     ...
22     fp = f1();
23     ...
24     fp();
25     return EXIT_SUCCESS;
26 }

```

Figure 2: **VCG Construction Example.** Without symbol information $V[VCG] = \{\text{main}, f1, f3, f6, f7, f8, f9, f10\}$, whereas with symbols (or debugging information) available, $V[VCG] = \{\text{main}, f1, f3, f9, f10\}$.

Figure 2 illustrates a C-like program, which we will be using as an example to demonstrate the VCG construction. `sysfilter` will initially include `main` (ln. 19) and `f9` (ln. 5). DCG will also include all the directly-reachable functions from the above initial set: `f1` (reachable from `main`, ln. 22) and `f10` (reachable from `f9`, ln. 22). Hence, $V[DCG] = \{\text{main}, f1, f9, f10\}$. Next, `sysfilter` will proceed with the construction of the ACG, which, initially, will include all the address-taken functions: `f3` (ln. 14), `f4` (ln. 13), and `f6` and `f7` (ln. 17). ACG will also include all the directly-reachable functions from set of AT functions: `f5` (reachable from `f4`, ln. 10) and `f8` (reachable from `f7`, ln. 7). Thus, $V[ACG] = \{f3, f4, f5, f6, f7, f8\}$.

`sysfilter` will then continue with pruning the ACG as follows. First, it will remove `f4`, as its address is only taken in function `f2` (ln. 13), which is unreachable. This will also result in removing `f5`, as it is only directly-reachable from `f4` (ln. 10). If the respective ELF object is stripped, the pruning process will terminate at this point, resulting in the following set of functions: $V[ACG'] = \{f3, f6, f7, f8\}$. If symbol (or debugging) information is available, then `sysfilter` can perform more aggressive pruning by identifying that `fp_arr` is not referenced by any function in scope. Therefore, the AT functions `f6` and `f7` can also be removed, as well as `f8` that is directly-reachable only from `f7` (ln. 7). The net result of the above is the following set of functions: $V[ACG''] = \{f3\}$.

To summarize, without symbol information, $V[VCG] = V[DCG] \cup V[ACG'] = \{\text{main}, f1, f3, f6, f7, f8, f9, f10\}$, whereas with symbols (or debugging information) available, $V[VCG] = V[DCG] \cup V[ACG''] = \{\text{main}, f1, f3, f9, f10\}$. Interested readers are referred to the appendix (§ A) for more information about how `sysfilter` handles GNU IFUNC and NSS symbols, overlapping code, and hand-written assembly.

3.1.3 System Call Set Construction

The x86-64 ABI dictates that system calls are performed using the `syscall` instruction [30].³ Moreover, during the invocation of `syscall`, the *system call number* is placed in register `%rax`. Armed with the program’s VCG, `sysfilter` constructs the system call set in question as follows.

First, it identifies all reachable functions that include `syscall` instructions, by performing a linear sweep in each function $f \in V[VCG]$ to pinpoint `syscall` instances. Once the set of all the reachable `syscall` instructions is established, `sysfilter` continues with performing a simple *value-tracking* analysis to resolve the exact value(s) of `%rax` on every `syscall` site. The process relies on standard live-variable analysis using *use-define* (UD) chains [2, § 9.2.5]. Specifically, `sysfilter` considers that `syscall` instructions “use” `%rax` and leverages the UD links to find all the instructions that “define” it. In most cases, `%rax` is defined via constant-load instructions (e.g., `mov $0x3, %eax`), and by collecting such instructions and extracting the respective constant values, `sysfilter` can assemble system call sets. If `%rax` is defined via instructions that involve memory operands, `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may be incomplete [72]. The output of the `syscall`-set extraction component is the collected set of system call numbers in JSON format (see Figure 1D).

We opt for applying the analysis above in an intra-procedural manner, as our results indicate that this strategy works well in practice (see § 5); system call invocation is architecture-specific, and typically handled via `libc` using the following pattern (in x86-64): `‘mov $SYS_NR, %eax; syscall’`, where `$SYS_NR = {0x0, 0x1, ...}`. One exception is the handling of the `syscall()` function [48], which performs system calls indirectly by receiving the respective system call number as argument. If `syscall()` is not-address taken in VCG, then `sysfilter` first identifies the reachable functions that directly-invoke `syscall()`, and performs intra-procedural, value-tracking on register `%rdi` (first argument, system call number). If the address of `syscall()` is taken in the reachable VCG, then `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may, again, be incomplete.

³Performing `syscalls` via software interrupts (e.g., `int $0x80`), or `sysenter`, is only supported in x86-64 Linux to allow executing 32-bit applications over a 64-bit kernel. `sysfilter` focuses solely on 64-bit applications (i.e., it does not consider `syscalls` via `int $0x80` or `sysenter`).

```

1 #define ARCH AUDIT_ARCH_X86_64
2 #define NRMAX (X32_SYSCALL_BIT - 1)
3 #define ALLOW SECCOMP_RET_ALLOW
4 #define DENY SECCOMP_RET_KILL_PROCESS
5
6 struct sock_filter filter[] = {
7 BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
8   (offsetof(struct seccomp_data, arch))),
9 BPF_JUMP(BPF_JMP | BPF_JEQ|BPF_K, ARCH, 0, 7),
10 BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
11   (offsetof(struct seccomp_data, nr))),
12 BPF_JUMP(BPF_JMP|BPF_JGT|BPF_K, NRMAX, 5, 0),
13 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0, 3, 0),
14 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1, 2, 0),
15 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 15, 1, 0),
16 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 60, 0, 1),
17 BPF_STMT(BPF_RET|BPF_K, ALLOW),
18 BPF_STMT(BPF_RET|BPF_K, DENY) };

```

Figure 3: **Classic BPF (cBPF) Program.** Compiled-by `sysfilter`, enforcing the following syscall set: 0 (read), 1 (write), 15 (exit), and 60 (sigreturn). The filter checks if the value of field `nr` (syscall number) $\in \{0, 1, 15, 60\}$ via means of linear search.

3.2 System Call Set Enforcement

The input to the syscall-set enforcement component of `sysfilter` is the set of allowed system calls, as well as the ELF file that corresponds to the main binary of the application (see Figure 1E). Armed with the set of developer-intended syscalls, `sysfilter` uses `seccomp-BPF` [36] to enforce it at run-time. The latter receives as input a BPF “program” [54], passed via `prctl`, or `seccomp`, which is invoked by the kernel on every system call. Note that BPF programs are executed in kernel mode by an interpreter for BPF bytecode [54], while just-in-time (JIT) compilation to native code is also supported [11]. In addition, the Linux kernel provides support for two different BPF variants: (a) classic (cBPF) and (b) extended (eBPF) [46]; `seccomp-BPF` makes use of cBPF only.

The input to `seccomp-BPF` programs (filters) is a fixed-size struct (i.e., `seccomp_data`; see Figure 8 in Appendix B), passed by the kernel, which contains a snapshot of the system call context: i.e., the syscall number (field `nr`), architecture (field `arch`), as well as the values of the instruction pointer and syscall arguments. `sysfilter` performs filtering based on the value of `nr` as follows: **if** ($nr \in \{0, 1, \dots\}$) **then** `ALLOW` **else** `DENY`, where $\{0, 1, \dots\}$ is the set of allowed system call numbers. Given such a set, `sysfilter` compiles a cBPF filter that implements the above check via means of *linear* or *skip list*-based search. Figure 3 depicts a filter that uses the linear search approach to enforce the following set of syscalls: read (0), write (1), exit (15), and sigreturn (60). Ln. 7 – 12 implement a standard preamble, which asserts that the architecture is indeed x86-64. This check is crucial as it guarantees that the mapping between the allowed syscall numbers and the syscalls performed is the right one.

For instance, suppose that this check is missing, and `getuid` (102)—a harmless syscall—exists in the allowed set. If the target process (x86-64) is compromised, and the attacker issues syscall no. 102, via `int $0x80` (or `sysenter`), then the filter will allow the syscall but the kernel will execute `socketcall` instead: i.e., the syscall with number 102 in x86 (32-bit), effectively giving the attacker network-access capabilities. The check in ln. 9 rejects every architecture different from x86-64, while the check in ln. 12 rejects syscalls that correspond to the x32 ABI [50].⁴ The bulk part of the enforcement/search is implemented in ln. 13 – 16 (BPF_JEQ statements). Note that cBPF does not allow loops, and therefore `sysfilter` implements the linear search using *loop unwinding* (i.e., ‘if-else if-...-else’ construct). In case of a non-permitted syscall, `sysfilter` terminates the processes (ln. 4, `SECCOMP_RET_KILL_PROCESS`). Figure 9, in the appendix (§ B), illustrates a cBPF filter that uses the skip list approach to implement the search.

`sysfilter` injects the compiled filter as follows. First, it generates a dynamic shared object, namely `libsysfilter.so`. Next, it links the aforementioned shared object with the main binary, using `patchelf` [61]; `libsysfilter.so` includes only a single function, `install_filter`, registered as a constructor. The net result of the above is that `ld.so` will automatically load `libsysfilter.so`, and invoke `install_filter`, during the initialization of the main binary (see Figure 1F).

`install_filter` attaches the compiled cBPF filter, at load-time, using the `seccomp` system call [47]. Importantly, before invoking `seccomp` (with `SECCOMP_SET_MODE_FILTER`), the `no_new_privs` attribute of the calling thread is asserted, via invoking `prctl` (with `PR_SET_NO_NEW_PRIVS`), disabling the acquisition of new privileges via further `execve`-ing programs that make use of `set-user-ID`, `set-group-ID`, or other capabilities. Lastly, `install_filter` passes the argument `SECCOMP_FILTER_FLAG_TSYNC` [47] to `seccomp` for making the respective filter visible to *all* executing threads, while it also uses `SECCOMP_FILTER_FLAG_SPEC_ALLOW` [47] to disable the speculative store bypass (SSB) mitigation. Note that the latter is configurable; however, the SSB mitigation is only relevant when BPF programs of *unknown provenance* are loaded in kernel space to further assist mounting Spectre attacks [37] (variant 4 [25])—`sysfilter` cBPF programs are not malicious nor attacker-controlled.

Once the filter is installed using the method outlined above, the respective process can execute only developer-intended syscalls. Note that `ld.so` is included in the analysis scope, and hence the initialization/finalization of additional libraries, at run-time (e.g., via `dlopen/dlclose`), as well as any other `ld.so`-related functionality, is supported seamlessly.

⁴If the target binary is going to be executed atop an x86-64 Linux kernel that does not support x86 emulation (`CONFIG_IA32_EMULATION=n`) nor the x32 ABI (`CONFIG_X86_X32=n`), then `sysfilter` can further optimize the generated filters by omitting the `arch`-related preamble.

Crucially, `no_new_privs` guarantees that filters are *pinned* to the protected process during its lifetime—i.e., even if the process is completely compromised, attackers cannot remove filters. Recall that the filtering itself takes place in kernel mode using only the syscall number as input; the syscall arguments are not inspected, and user space memory is not accessed, thereby avoiding the pitfalls related to concurrency and (wrapper-based) syscall filtering [17, 92]. In addition, applications that make use of `seccomp-BPF` are seamlessly supported as well. BPF filters are *stackable*, meaning that more than one filter can be attached to a process; if multiple filters exist, the kernel always enforces the most *restrictive* action.

Handling `execve` `sysfilter` prevents enforcement by-passes via the execution of different programs. Specifically, even if a (compromised) process is allowed to invoke `execve`, it still cannot extend its set of allowed syscalls by invoking a different executable that has a (potentially) larger set of allowed syscalls; the same is also true if the process tries to craft a rogue executable in the filesystem, which allows all syscalls (or some of the blocked ones), and execute it. Filter pinning and stacking are essential for ensuring that processes can only reduce their set of allowed syscalls, in accordance to the principle of least privilege [76], but they do interfere with `execve` as they are preserved process attributes.

For example, suppose that programs `P1` and `P2` have the following syscall sets. `P1`: 0 (read), 1 (write), 15 (exit), and 59 (`execve`); `P2`: 0 (read), 1 (write), 2 (open), 3 (close), 8 (lseek), 9 (mmap), 11 (munmap), 56 (clone), 61 (wait4), 79 (getcwd), 96 (gettimeofday), 102 (getuid), 115 (getgroups), 202 (futex), 292 (dup3), and 317 (`seccomp`). If `P2` is normally-invoked, then it will operate successfully. However, if `P2` is invoked via `P1`, then the resulting process will not be able to issue any other syscall than `read`, `write`, `exit`, and `execve` (the last two are not even required by `P2`). To deal with this issue, `sysfilter` supports two different `execve` modes: (a) *union* and (b) *hierarchical*.

In union mode, given a set programs $\{P_1, P_2, \dots, P_N\}$ that can be invoked in any combination, via `execve`, with $SYS_1, SYS_2, \dots, SYS_N$ being their allowed sets of syscalls, `sysfilter` constructs a filter that enforces the union of $SYS_1, SYS_2, \dots, SYS_N$ and attaches it to all of them. This will result in each process functioning correctly, as it has support for the syscalls it requires, but overly-approximates least privilege—every program effectively inherits the privileges (with respect to the syscall API) of all others in the set. In the example above, union would result in executing both `P1` and `P2` with the following syscall set: $\{0 - 3, 8, 9, 11, 15, 56, 59, 61, 79, 96, 102, 115, 202, 292, 317\}$.

In hierarchical mode, `sysfilter` begins with the same approach as above, but further *rectifies* (reduces) syscall sets each time a process invokes `execve`. In our example, this would result in executing `P1` with the set $\{0 - 3, 8, 9, 11, 15, 56, 59, 61, 79, 96, 102, 115, 202, 292, 317\}$, further reduced to $\{0 - 3, 8, 9, 11, 56, 61, 79, 96, 102, 115, 202,$

$292, 317\}$ right before `execve`-ing `P2`. Note that the hierarchical mode still results in certain processes being a bit more privileged (with respect to accessing OS services), but not all.

If special treatment regarding `execve` is required for a particular (set of) program(s), then a *recipe* can be provided to the enforcement tool, along with the respective syscall sets, which describes how `sysfilter` should operate on `execve` calls. If union mode is specified, then `sysfilter` merely splices together a set of different syscall sets (provided via separate JSON files), and compiles a single filter that is attached to all programs in scope. In case of hierarchical mode, the recipe describes the (`execve`) relationships between callers and callees, allowing `sysfilter` to construct different filters, one for each program in scope, which adhere to the model above.

More importantly, our results (§ 5) indicate that recipe creation can be automated, to some extent, by employing static value-tracking analysis to resolve the first argument of `execve` calls. However, note that `sysfilter` is not geared towards sandboxing applications that invoke *arbitrary* scripts or programs (e.g., command-line interpreters, managed runtime environments); other schemes, like Hails [21], SHILL [56], and the Web API Manager [82], are better suited for this task.

3.3 Prototype Implementation

Our prototype implementation consists of ≈ 2.5 KLOC of C/C++ and ≈ 150 LOC of Python, along with various shell scripts (glue code; ≈ 120 LOC). More specifically, we implemented the extraction tool atop the Egalito framework [95]. Egalito is a binary *recompiler*; it allows rewriting binaries in-place by first lifting binary code into a layout-agnostic, machine-specific intermediate representation (IR), dubbed EIR, and then allowing “tools” to inspect or alter it.

We implemented the extraction tool as an Egalito “pass” (C/C++), which creates the analysis scope, constructs the VCG, and extracts the respective syscall set, using the techniques outlined in § 3.1. Note that we do not utilize the binary rewriting features of Egalito; we only leverage the framework’s API to precisely disassemble the corresponding binaries and lift their code in EIR form, which, in turn, we use for implementing the analyses required for constructing the DCG, identifying all AT functions for building the ACG, pruning unreachable parts of the call graph for assembling the VCG, identifying `syscall` instructions, performing value-tracking, *etc.* We chose Egalito over similar frameworks as it employs the best jump table analysis to date.

The enforcement tool is implemented in Python and is responsible for generating the `cBPF` filter(s), and `libsfilter.so`, and attaching the latter to the main binary using `patchelf` (see § 3.2). Lastly, during the development of `sysfilter` we also improved Egalito by adding better support for hand-coded assembly, fixing various symbol resolution issues, and re-architecting parts of the framework to reduce memory pressure. (We upstreamed all our changes.)

Application	Version	Syscalls	Tests	Pass?	
FFmpeg	(124)	4.2	167	3756	✓
GNU Core.	(100)	8.31	148	672	✓
GNU M4	(1)	1.14	70	236	✓
MariaDB	(156)	10.3	153	2059	✓
Nginx	(1)	1.16	128	356	✓
Redis	(6)	5.0	104	81	✓
SPEC CINT.	(12)	1.2	82	12	✓
SQLite	(7)	3.31	139	31190	✓
Vim	(3)	8.2	163	255	✓
GNU Wget	(1)	1.20	107	130	✓

Table 1: **Correctness Test.** The numbers in parentheses count the different binaries included in the application/package. “Syscalls” indicates the number of system calls in the allowed set; in case of applications with multiple binaries that number corresponds to the unique syscalls across the syscall sets of all binaries in the package. “Tests” denotes the number of tests run from the validation suite of the application.

4 Evaluation

We evaluate `sysfilter` in terms of (1) correctness, (2) run-time performance overhead, and (3) effectiveness.

Testbed We used two hosts for our experiments: (a) run-time performance measurements were performed on an 8-core Intel Xeon W-2145 3.7GHz CPU, armed with 64GB of (DDR4) RAM, running Devuan Linux (v2.1, kernel v4.16); (b) analysis tasks were performed on an 8-core AMD Ryzen 2700X 3.7GHz CPU, armed with 64GB of (DDR4) RAM, running Arch Linux (kernel v5.2). All applications (except SPEC CINT2006) were obtained from Debian `sid` (development distribution) [86], as it provides the latest versions of upstream packages along with debug/symbol information [93].

Correctness We used 411 binaries from various packages/projects with `sysfilter`, including GNU Coreutils, Nginx, Redis, SPEC CINT2006, SQLite, FFmpeg, MariaDB, Vim, GNU M4, and GNU Wget, to extract and enforce their corresponding syscall sets. The results are shown in Table 1. Once sandboxed, we stress-tested them with $\approx 38.5K$ tests from the projects’ validation suites. (Note that we did not include tests that required the application to execute arbitrary external programs, such as tests with arbitrary commands used in Vim scripts, Perl scripts in Nginx, and arbitrary shell scripts to load data in SQLite and M4.) In all cases, `sysfilter` managed to extract a complete and tight over-approximation of the respective syscall sets, demonstrating that our prototype can successfully handle complex, real-world software.

Performance To assess the run-time performance impact of `sysfilter`, we used SPEC CINT2006 (std. benchmarking suite), Nginx (web server), and Redis (data store)—i.e., 19 program binaries in total; the selected binaries represent the most performance-sensitive applications in our set and are well-suited for demonstrating the relative overhead(s). We

also explored different settings and configurations, including interpreted vs. JIT-compiled BPF filters, and filter code that implements sandboxing using a linear search vs. filter code that utilizes a skip list-based approach (§ 3.2). In the case of SPEC, we observed a run-time slowdown $\leq 1\%$ under all conditions and search methods.

Figure 4 and Figure 5 illustrate the impact of `sysfilter` on Nginx (128) and Redis (103)—the numbers in parentheses indicate the corresponding syscall set sizes, while “Binary” corresponds to skip list-based filters. We configured Nginx to use 4 worker processes and measured its throughput using the `wrk` tool [23], generating requests via the loopback interface from 4 threads, over 256 simultaneous connections, for 1 minute. Overall, `sysfilter` diminishes reduction in throughput by using skip list-based filters (compared to linear search-based ones) when JIT is disabled, with maximum reductions in throughput of 18% and 7%, respectively. The differences in compilation strategy appear to be normalized by jitting, which showed a maximum drop in throughput of 6% in all conditions. We evaluated Redis similarly, using the `memtier` tool [73], performing a mix of SET and GET requests with a 1:10 req. ratio for 32 byte data elements. The requests were issued from 4 worker threads with 128 simultaneous connections, per thread, for 1 minute. `sysfilter` incurs maximum throughput reductions of 11% and 3%, with and without JIT support, respectively. Lastly, toggling `SECCOMP_FILTER_FLAG_SPEC_ALLOW` [47] (for enabling the SSB mitigation) incurs an additional $\approx 10\%$ overhead in all cases.

Effectiveness To assess the security impact of syscall filtering, we investigated how `sysfilter` reduces the attack surface of the OS kernel, by inquiring what percentage of all C/C++ applications in Debian `sid` ($\approx 30K$ main binaries) can exploit 23 (publicly-known) Linux kernel vulnerabilities—ranging from memory disclosure and corruption to direct privilege escalation—even after hardened with `sysfilter`. A list of the number of binaries in our dataset affected by each CVE is shown in Table 2. Depending on the exact vulnerability, the percentage of binaries that can still attack the kernel ranges from 0.09% – 64.34%. Although `sysfilter` does not defend against particular types of attacks (e.g., control- or data-flow hijacking [87]), our results demonstrate that it can mitigate real-life threats by means of least privilege enforcement and (OS) attack surface reduction.

5 Large-scale System Call Analysis

We conclude our work with a large scale study regarding the syscall profile of all C/C++ applications in Debian `sid`, reporting insights regarding syscall set sizes (e.g., the number of syscalls per binary), most- and least-frequently used syscalls, syscall site distribution (libraries vs. main binary), and more. The results of this analysis not only justify our design, but can also aid that of future syscall policing mechanisms.

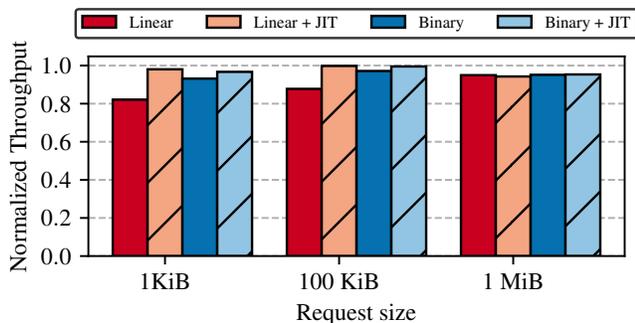


Figure 4: Impact of `sysfilter` on Nginx.

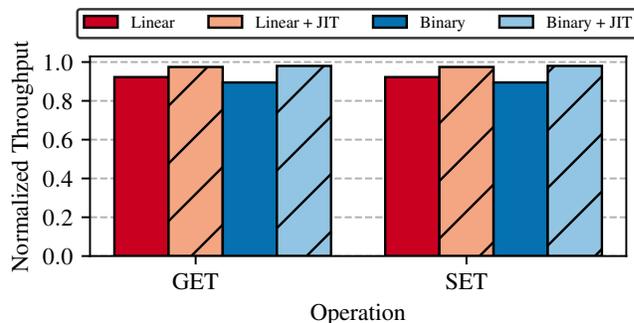


Figure 5: Impact of `sysfilter` on Redis.

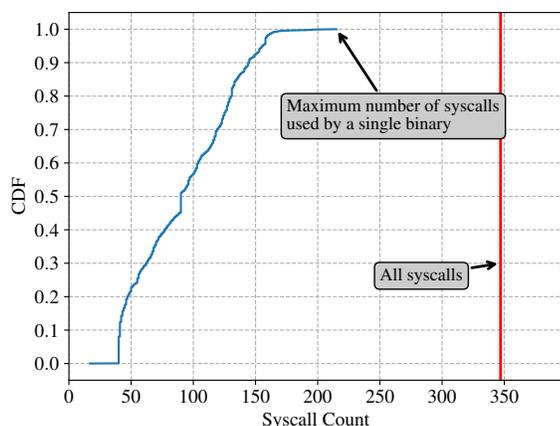


Figure 6: Number of Syscalls per Binary.

We consider packages from the three major Debian repositories, namely `main`, `contrib`, and `nonfree`. At the time of writing, the Debian `sid` distribution contains over 50K packages listed for the `x86-64`. We exclude, however, packages that do not contain executable code, such as documentation packages (`*-doc`), development headers (`*-dev`), and debug symbols (`*-dbg`, `*-dbgsym`); meta-packages and virtual packages; architecture-agnostic packages that do not include `x86-64` binaries; and packages containing only shared libraries. Note that shared libraries and other excluded packages can be installed during processing, as dependencies of main application packages. We processed a total of 33829 binaries across 8922 packages, 30398 (91.3%) of which could be analyzed successfully. The median runtime for the extraction tool is about 30s per binary, with 90% of binaries completing within 200s. For a single FCG pass, the median runtime is reduced to about 10s per binary.

Number of Syscalls per Binary Many binaries only use a small portion of the syscall API. Figure 6 shows the distribution of the number of syscalls used by each binary processed.

Out of the total set of 347 syscalls, the maximum number of syscalls used by one binary is only 215 syscalls, which represents 62% of the total syscall API. We observe that this distribution has a long tail: i.e., the median syscall count per binary is 90 syscalls, with the 90th percentile at 145 syscalls.

With one exception, we find that *all* binaries processed use at least 40 syscalls. The sole exception is the statically-linked binary `mtcp_restart` that uses only 17 syscalls—this binary performs syscalls directly, without using any library wrappers. In the general case, even the simplest of programs, such as `/bin/false`, utilize 40 syscalls due to the paths included by initialization functions that load shared libraries: e.g., `mmap` and `mprotect` are ubiquitous as they are always reachable from `_start`, even before `main` is invoked.

Syscalls from Libraries When extracting syscalls from each binary, we record which of its shared libraries contained a syscall instruction for each invocation. After `libc`, the second most common library is `libpthread` (used by 68.7% of binaries), in which we observe 40 syscalls—upon further investigation we found that this is due to `glibc`'s use of macros to directly invoke syscalls. The next most common libraries that invoke syscalls directly are `libstdc++` (37% of binaries), which invokes `futex` (202) directly; `libnss_resolve` (32% of binaries, 3 syscalls), and `libglib-2.0` (17%, 5 syscalls).

While we leave a comprehensive analysis of library syscalls to a future study, we note a few common themes. Libraries tend to directly invoke syscalls with no `libc` wrapper function available, such as `futex`, and `gettid`. We also observe many libraries directly invoking syscalls specific to their core functionality, perhaps to handle circumstances where existing wrappers are unavailable or insufficient. For instance, we note cryptographic libraries, such as `libcrypt` and `libcrypto`, directly invoking cryptographic-related syscalls, like `getrandom` and `keyctl`.

Effectiveness of FCG Approximation Figure 7 shows the number of syscalls we extract from each binary using the three FCG approximation methods (§ 3.1.2): `DCG`, `DCG ∪ ACG`, and `VCG`, sorted by the count for the `VCG`. Each binary represents three points on the figure (i.e., one for each method).

CVE	Syscall(s) Involved	Vulnerability Type	Binaries (%)
CVE-2019-11815	clone, unshare	Memory corruption	19558 (64.34)
<u>CVE-2013-1959</u>	write	Direct privilege escalation	19558 (64.34)
<u>CVE-2015-8543</u>	socket	Type confusion	19128 (62.93)
<u>CVE-2017-17712</u>	sendto, sendmsg	Memory corruption	19057 (62.69)
<u>CVE-2013-1979</u>	recvfrom, recvmsg	Direct privilege escalation	18968 (62.40)
<u>CVE-2016-4998</u>	setsockopt	Memory disclosure	17360 (57.11)
<u>CVE-2016-4997</u>	setsockopt	Memory corruption	17360 (57.11)
<u>CVE-2016-3134</u>	setsockopt	Memory corruption	17360 (57.11)
<u>CVE-2017-18509</u>	setsockopt, getsockopt	Memory corruption	17416 (57.29)
CVE-2018-14634	execve, execveat	Memory corruption on suid program	16775 (55.18)
CVE-2017-14954	waitid	Memory disclosure	14064 (46.27)
<u>CVE-2014-5207</u>	mount	Direct privilege escalation	11412 (37.54)
CVE-2018-12233	setxattr	Memory corruption	3356 (11.04)
CVE-2016-0728	keyctl	Memory corruption	2827 (9.30)
CVE-2014-9529	keyctl	Memory corruption	2827 (9.30)
CVE-2019-13272	ptrace	Direct privilege escalation	127 (0.42)
CVE-2018-1000199	ptrace	Memory corruption	127 (0.42)
CVE-2014-4699	fork, clone, ptrace	Register value corruption	121 (0.40)
<u>CVE-2014-7970</u>	pivot_root	DoS	79 (0.26)
CVE-2019-10125	io_submit	Memory corruption	58 (0.19)
CVE-2017-6001	perf_event_open	Direct privilege escalation	51 (0.17)
CVE-2016-2383	bpf	Memory corruption	35 (0.12)
CVE-2018-11508	adjtimex	Memory disclosure	26 (0.09)

Table 2: **Effectiveness Analysis.** The column “Binaries” indicates the number (and percentage) of binaries observed in the large scale analysis on Debian `suid` applications that use the system calls related to the respective vulnerability. (Underlined entries correspond to vulnerabilities that involve namespaces.)

For all binaries, the count for the VCG is always in between that of DCG and $DCG \cup ACG$. Thus, for our dataset, VCG represents a safe, *tight* over-approximation of the FCG.

dl{open, sym} and execve By employing our value-tracking approach (see § 3.1.3), `sysfilter` can resolve $\approx 89\%$ of all `dlsym` arguments, $\approx 37\%$ of all `dlopen` arguments, and $\approx 30\%$ of all `execve` arguments. We observed a few cases in common libraries where value-tracking fails, which may benefit from special handling (§ A): e.g., $\approx 50\%$ of `dlsym` failures relate to NSS functionality, while $\approx 5\%$ of `dlopen` failures involve Kerberos plugins. Lastly, we found two isolated cases where `sysfilter` was unable to construct syscall sets: `Qemu` and `stress-ng` contain arbitrary syscall dispatchers (like `glibc`’s `syscall()`), which is expected given their functionality. Otherwise, we find that syscall sites follow strictly the pattern `mov $SYS_NR, %eax; syscall`.

6 Related Work

Syscall-usage Analysis Tsai et al. [88] performed a study similar to ours (on binaries in Ubuntu v15.04) to characterize the usage of the syscall API, as well as that of `ioctl`, `fcntl`, `prctl`, and pseudo-filesystem APIs. Their study focuses on quantifying API complexity and security-related usage trends, such as unused syscalls and adoption of secure

APIs over the legacy ones. Our study focuses specifically on the syscall API as a means of evaluating our extraction tool. We consider this work complementary, and focus on making the analysis more scalable, precise, and complete. Specifically, and in antithesis to `sysfilter`, the call graph construction approach of Tsai et al. does not consider initialization/finalization code nor does it identify AT functions that are part of global `struct/union/C++` object initializers.

Static System Call Filtering Syscall filtering has been extensively studied in the past, in various contexts. Indeed, `sysfilter` shares many of the problems, and proposed solutions, with the seminal work by Wagner and Dean [89], which uses static analysis techniques to model sequences of valid syscalls as a non-deterministic finite automaton (NFA). This work, as well as others from its era [22], aim at building models of program execution for intrusion detection purposes. In contrast, `sysfilter` focuses on building optimized (OS-enforceable) `seccomp-BPF` filters by determining the total set of syscalls, independent of ordering, which provides a more compact representation and eliminates the challenges related to control flow modeling. Moreover, `sysfilter` employs binary analysis, whereas Wagner and Dean’s work requires recompiling target binaries and shared libraries, which severely limits the deployability of their scheme.

Shredder [55] performs static analysis on Windows binaries to identify API calls, and arguments, used by applications.

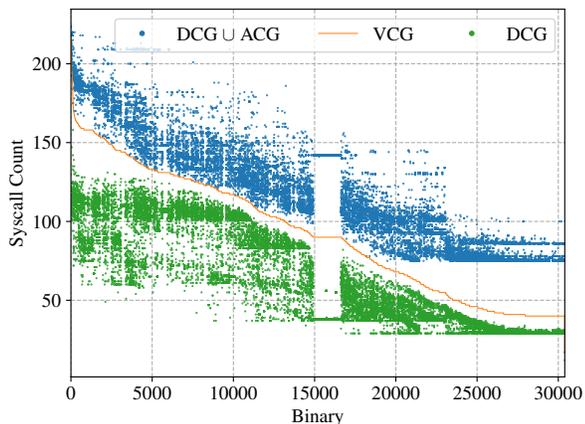


Figure 7: **Syscall Count for Different FCG Construction Methods.** The number of syscalls reported for each binary is shown, sorted by the count for the VCG.

Specifically, it restricts calls to syscall wrapper functions, via trampolines, but requires CFI for effective protection. Independently and concurrently to our work, Ghavamnia et al. proposed Confine [20]: a (mostly) static analysis-based system for automatically extracting and enforcing syscall policies on “containerized” (i.e., Docker) applications. Confine requires access to C library code (e.g., `glibc` or `musl`), while its call graph construction approach considers every function in non-`libc` code within scope. In addition, it relies on `objdump`, and hence, requires symbols for precise disassembly. `sysfilter` can operate on stripped binaries, while our FCG construction approach produces much tighter syscall sets.

Similar to `sysfilter`, Zeng et al. [98] identify valid sets of syscalls using binary analysis, but their approach lacks soundness: its call graph approximation method relies, in part, on points-to analysis to resolve the targets of function pointers. In antithesis, `sysfilter` identifies all address-taken functions in order to avoid the impression issues associated with this method. Further, Zeng et al. perform the enforcement using a customized Linux kernel to provide per-process system call tables, whereas our `seccomp-BPF` based approach is available in stock Linux kernel v3.5 or later.

Dynamic System Call Filtering Systrace [69] uses dynamic tracing to generate system call policies and implements a userspace daemon for enforcement. Mutz et al. [58] and Maggi et al. [51] develop statistical models for host-based intrusion detection, which as a design choice inherently gives false negatives, potentially impeding valid program execution. Ostia [18] provides a system call sandboxing mechanism that delegates policy decisions to per-process agents, while a plethora of earlier work on container debloating [90], and sandboxing [42, 90], also relies on dynamic tracing. In contrast, `sysfilter` does not rely on dynamic syscall tracing or statistical models, which can generate incomplete policies—

instead, `sysfilter` safely over-approximates a program’s true syscall set and thus will not break program execution.

seccomp-BPF in Existing Software Firefox [57], Chrome [8], and OpenSSH [63] use `seccomp-BPF` to sandbox themselves using manually-crafted policies, while container runtimes, such as Docker and Podman, allow the use `seccomp-BPF` policies to filter container syscalls. By default, Docker applies a filter that disables 44 syscalls [15], and Podman has support for tracing syscalls dynamically with `ptrace` to build a profile for containers. Both also fully support user-specified filters [75]. `sysfilter` can be seamlessly integrated with such software, providing the apparatus for generating the respective syscall sets automatically/precisely.

Binary Debloating `sysfilter` shares goals and analysis approaches with recent software debloating techniques. Quach et al. [71] propose a compiler-based approach that embeds dependency information into programs, and uses a custom loader to selectively load only required portions of shared libraries in memory. TRIMMER [80] specializes LLVM bytecode based on a user-defined configuration, while the work of Koo et al. [39] utilizes coverage information to remove code based on feature directives. C-Reduce [74], Perses [85], and CHISEL [27] use delta-debugging techniques to compile minimized programs using a series of provided test cases. Unlike previous approaches, Razor [70] does not require source code, and implements a dynamic tracer to reconstruct the program’s FCG from a set of test cases. The analysis used by Nibbler [1] is the most similar to `sysfilter`. However, Nibbler requires symbols, whereas `sysfilter` operates on stripped binaries.

7 Conclusion

We presented `sysfilter`: a static (binary) analysis-based framework that automatically limits what OS services attackers can (ab)use, by enforcing the principle of least privilege, and reduces the attack surface of the OS kernel, by restricting the syscall set available to userland processes. We introduced a set of program analyses for constructing syscall sets in a scalable, precise, and complete manner, and evaluated our prototype in terms of correctness using 411 binaries, from various real-world C/C++ projects, and $\approx 38.5K$ tests to stress-test their functionality when armored with `sysfilter`. Moreover, we assessed the impact of our syscall enforcement mechanism(s) using SPEC CINT2006, Nginx, and Redis, demonstrating minimal run-time slowdown. Lastly, we concluded with a large scale study about the syscall profile of $\approx 30K$ C/C++ applications (Debian `sid`). We believe that `sysfilter` is a practical and robust tool that provides a solution to the problem of unlimited access to the syscall API.

Availability

The prototype implementation of `sysfilter` is available at: <https://gitlab.com/brown-ssl/sysfilter>

Acknowledgments

We thank our shepherd, Aravind Prakash, and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) and the Defense Advanced Research Projects Agency (DARPA) through awards N00014-17-1-2788 and HR001118C0017. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, ONR, or DARPA.

References

- [1] Ioannis Agadakis, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (AC-SAC)*, pages 70–83, 2019.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd edition, 2006.
- [3] Jim Alves-Foss and Jia Song. Function Boundary Detection in Stripped Binaries. In *Annual Computer Security Applications Conference (AC-SAC)*, pages 84–96, 2019.
- [4] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium (SEC)*, pages 583–600, 2016.
- [5] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189, 2017.
- [6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security Symposium (SEC)*, pages 845–860, 2014.
- [7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *IEEE Symposium on Security and Privacy (S&P)*, pages 227–242, 2014.
- [8] Chromium Blog. A safer playground for your linux and chrome os renderers. <https://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html>.
- [9] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.
- [10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 559–572, 2010.
- [11] Jonathan Corbet. BPF: the universal in-kernel virtual machine. <https://lwn.net/Articles/599755/>.
- [12] Gabriel Corona. The ELF file format. <https://www.gabriel.urdhr.fr/2015/09/28/elf-file-format/>.
- [13] Solar Designer. Getting around non-executable stack (and fix). <https://seclists.org/bugtraq/1997/Aug/63>.
- [14] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 128–142, 2020.
- [15] Docker Documentation. Seccomp Security Profiles for Docker. <https://docs.docker.com/engine/security/seccomp/>.
- [16] Common Weakness Enumeration. CWE-123: Write-what-where Condition. <https://cwe.mitre.org/data/definitions/123.html>.
- [17] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed System Security Symposium (NDSS)*, pages 163–176, 2003.
- [18] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Network and Distributed System Security Symposium (NDSS)*, 2004.
- [19] Jason Geffner. VENOM: Virtualized Environment Neglected Operations Manipulation. <http://venom.crowdstrike.com>.
- [20] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.
- [21] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60, 2012.
- [22] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting Manipulated Remote Call Streams. In *USENIX Security Symposium (SEC)*, pages 61–79, 2002.
- [23] Will Glozer. wrk – a HTTP benchmarking tool. <https://github.com/wg/wrk>.
- [24] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, 2014.
- [25] Google Project Zero. speculative execution, variant 4: speculative store bypass. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1528>.
- [26] LLVM Developer Group. The LLVM Compiler Infrastructure. <https://llvm.org>.
- [27] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 380–394, 2018.
- [28] Gerard J. Holzmann. Code Inflation. *IEEE Software*, (2):10–13, 2015.
- [29] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 969–986, 2016.
- [30] Intel. System V Application Binary Interface. <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>.
- [31] GNU Compiler Collection (GCC) Internals. Common Function Attributes. <https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes>.
- [32] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1868–1882, 2018.
- [33] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.
- [34] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*, pages 957–972, 2014.
- [35] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user attacks. In *USENIX Security Symposium (SEC)*, pages 459–474, 2012.

- [36] The Linux Kernel. Seccomp BPF (SECure COMPUting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.
- [37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1–19, 2019.
- [38] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 461–477, 2018.
- [39] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *European Workshop on Systems Security (EuroSec)*, pages 1–6, 2019.
- [40] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.
- [41] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 276–291, 2014.
- [42] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-Phase Execution of Application Containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.
- [43] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *USENIX Annual Technical Conference (ATC)*, pages 1–13, 2017.
- [44] Percy Liang and Mayur Naik. Scaling Abstraction Refinement via Pruning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 590–601, 2011.
- [45] The GNU C Library. System Databases and Name Service Switch. https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html.
- [46] Linux Programmer’s Manual. bpf – perform a command on an extended BPF map or program. <http://man7.org/linux/man-pages/man2/bpf.2.html>.
- [47] Linux Programmer’s Manual. seccomp – operate on Secure Computing state of the process. <http://man7.org/linux/man-pages/man2/seccomp.2.html>.
- [48] Linux Programmer’s Manual. syscall – indirect system call. <http://man7.org/linux/man-pages/man2/syscall.2.html>.
- [49] Generic Part Linux Standard Base Core Specification. Exception Frames. https://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html.
- [50] H. J. Lu and Mike Frysinger. x32 System V Application Binary Interface. <https://sites.google.com/site/x32abi/>.
- [51] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 7(4):381–395, 2008.
- [52] Linux Programmer’s Manual. syscalls – Linux system calls. <http://man7.org/linux/man-pages/man2/syscalls.2.html>.
- [53] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 941–951, 2015.
- [54] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*, pages 259–270, 1993.
- [55] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Annual Computer Security Applications Conference (ACSAC)*, pages 1–16, 2018.
- [56] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 183–199, 2014.
- [57] MozillaWiki. Security/Sandbox/Seccomp. <https://wiki.mozilla.org/Security/Sandbox/Seccomp>.
- [58] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security (TISSEC)*, pages 61–93, 2006.
- [59] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.
- [60] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 31–40, 2010.
- [61] NixOS. patchelf – A small utility to modify the dynamic linker and RPATH of ELF executables. <https://github.com/NixOS/patchelf>.
- [62] A. Jefferson Offutt and J. Huffman Hayes. A Semantic Model of Program Faults. *ACM SIGSOFT Software Engineering Notes (SEN)*, 21(3):195–200, 1996.
- [63] OpenSSH. Release Notes. <https://www.openssh.com/txt/release-6.0>.
- [64] Oracle Solaris, Linker and Libraries Guide. Position-Independent Code. https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html.
- [65] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*, pages 420–436, 2017.
- [66] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. Kernel Protection against Just-In-Time Code Reuse. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1–28, 2019.
- [67] GNU Project. The GNU Compiler Collection. <https://gcc.gnu.org>.
- [68] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 584–598, 2020.
- [69] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium (SEC)*, pages 257–272, 2003.
- [70] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium (SEC)*, pages 1733–1750, 2019.
- [71] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium (SEC)*, pages 869–886, 2018.
- [72] Ganesan Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
- [73] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

- [74] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-Case Deduction for C Compiler Bugs. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.
- [75] Valentin Rothberg. Generate SECCOMP Profiles for Containers Using Podman and eBPF. <https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html>.
- [76] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *IEEE*, 63(9):1278–1308, 1975.
- [77] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 448–459, 2016.
- [78] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*, pages 745–762, 2015.
- [79] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.
- [80] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zafar. TRIMMER: Application Specialization for Code Debloating. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 329–339, 2018.
- [81] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 574–588, 2013.
- [82] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don’t Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 179–194, 2017.
- [83] Brad Spengler. PaX: The Guaranteed End of Arbitrary Code Execution. In *G-Con2*, 2003.
- [84] Guy Lewis Steele Jr. Debunking the “Expensive Procedure Call” Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *ACM National Conference*, pages 153–162, 1977.
- [85] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-Guided Program Reduction. In *International Conference on Software Engineering (ICSE)*, pages 361–371, 2018.
- [86] Debian The Universal Operating System. The unstable distribution (“sid”). <https://www.debian.org/releases/sid/>.
- [87] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2013.
- [88] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support When You’re Supporting. In *European Conference on Computer Systems (EuroSys)*, pages 1–16, 2016.
- [89] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, pages 156–168, 2000.
- [90] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.
- [91] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, pages 522–536, 2011.
- [92] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2007.
- [93] Debian Wiki. Using Symbols Files. <https://wiki.debian.org/UsingSymbolsFiles>.
- [94] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–382, 2016.
- [95] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, 2020.
- [96] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *USENIX Security Symposium (SEC)*, pages 1805–1821, 2019.
- [97] Yves Younan, Wouter Joosen, and Frank Piessens. Runtime Countermeasures for Code Injection Attacks against C and C++ Programs. *ACM Computing Surveys (CSUR)*, 44(3):1–28, 2012.
- [98] Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. Tailored Application-specific System Call Tables. Technical report, Pennsylvania State University, 2014.
- [99] Hanqing Zhao, Yanyu Zhang, Kun Yang, and Taesoo Kim. Breaking Turtles All the Way Down: An Exploitation Chain to Break out of VMware ESXi. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

A VCG Special Cases

GNU IFUNC GCC, along with the GNU Binutils and `glibc`, provide support for (a GNU-specific feature, named) *indirect functions*. Such symbols are of type `STT_GNU_IFUNC` and have an associated *resolver* function that will “return” the actual/target function to be used in lieu of the indirect (IFUNC) symbol [31]. The resolution takes place via PLT, at run-time; IFUNCs are typically used for customizing the symbol resolution of `ld.so`, and selecting among different function implementations that use processor-specific features. `sysfilter` links every call via an IFUNC PLT entry with: (1) the respective resolver function, and (2) *all* its potential targets in VCG—the latter are easily identifiable as they are functions whose address is taken in the resolver.

GNU NSS The Name Service Switch (NSS) [45] is used by `glibc` to select among different *name resolution* mechanisms (e.g., flat-file databases, DNS, LDAP). Specifically, `glibc` consults `nsswitch.conf` to determine the mapping between various databases (i.e., `passwd`, `shadow`, `group`, `hosts`, *etc.*) and resolution mechanisms (e.g., `files`, `dns`, `ldap`). Each such mechanism corresponds to a different dynamic shared

object (e.g., `libnss_files.so`, `libnss_dns.so`, `libnss_ldap.so`), which provides a specific implementation of the NSS API. Depending on the contents of `nsswitch.conf`, `glibc` loads the analogous `.so` ELF file, using `dlopen`, and invokes the relevant functions (NSS), after obtaining their addresses via `dlsym`. `sysfilter` parses `nsswitch.conf`, and makes use of the implicit library/function dependency mechanism to add the matching ELF object(s) and function(s) in the analysis scope (§ 3.1.1) and VCG (§ 3.1.2), respectively.

Overlapping Functions Certain versions of `glibc` include functions whose body overlaps with that of other functions. In particular, in v2.24 of `glibc`, ≈ 30 functions are completely embedded inside others (e.g., `connect` wraps `__connect_nocancel`). In cases where, say, `f1()` overlaps with `f2()`, and `&f1() < &f2()`, both functions can be (in)directly invoked by others, but if `f1()` gets executed, `f2()` will be invoked as well, as the execution will fall through to the latter. `sysfilter` supports such cases by carefully inspecting function boundaries (`.eh_frame` section; § 3.1.2), and connecting the respective functions in DCG, accordingly.

Hand-written Assembly ASM code is not problematic for `sysfilter` as long as it adheres to our hardening assumptions (§ 2). `sysfilter` does not support non-PIC objects (§ 3.1.1); hence, if ASM code that is embedded in binaries is analyzed, it will be PIC (by construction). If `.eh_frame` records (for hand-written ASM) are missing, or code and data are mixed, then the precision of our analyses will be affected. Thankfully, however, the (hand-written) ASM code that is linked-with popular binaries/libraries oftentimes contains annotations to support stack unwinding and (C++) exception handling [49]. Lastly, if partial information regarding function boundaries is available, `sysfilter` will resort to using a combination of linear and recursive disassembly techniques, and state-of-the-art heuristics [94], for approximating function boundaries.

B Enforcement Details

```
1 struct seccomp_data {
2     int nr;           /* syscall number */
3     __u32 arch;      /* architecture (x86-64) */
4     __u64 instruction_pointer; /* IP value */
5     __u64 args[6];  /* syscall arguments */
6 };
```

Figure 8: `struct seccomp_data`. Passed by the Linux kernel to `seccomp-BPF` filters on every syscall. The field `nr` is filled with the system call number, while `arch` and `instruction_pointer` are filled with the respective architecture, and value of the instruction pointer, during the time of executing syscall (i.e., `AUDIT_ARCH_X86_64` and `%rip`, in x86-64). Likewise, `args[6]` is a six-element array, filled with the syscall arguments (i.e., the values of registers `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, and `%r9`, in x86-64).

```
1 #define ARCH    AUDIT_ARCH_X86_64
2 #define NRMAX  (X32_SYSCALL_BIT - 1)
3 #define ALLOW  SECCOMP_RET_ALLOW
4 #define DENY   SECCOMP_RET_KILL_PROCESS
5
6 struct sock_filter filter[] = {
7     BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
8             (offsetof(struct seccomp_data, arch))),
9     BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, ARCH, 0, 7),
10    BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
11            (offsetof(struct seccomp_data, nr))),
12    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 61, 11, 0),
13    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 8, 5, 0),
14    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 2, 2, 0),
15    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 1, 19, 0),
16    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 0, 18, 19),
17    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 3, 17, 0),
18    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 2, 16, 17),
19    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 11, 2, 0),
20    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 9, 14, 0),
21    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 8, 13, 14),
22    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 56, 12, 0),
23    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 11, 11, 12),
24    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 115, 5, 0),
25    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 96, 2, 0),
26    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 79, 8, 0),
27    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 61, 7, 8),
28    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 102, 6, 0),
29    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 96, 5, 6),
30    BPF_JUMP(BPF_JMP | BPF_JGE | BPF_K, 292, 2, 0),
31    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 202, 3, 0),
32    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 115, 2, 3),
33    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 317, 1, 0),
34    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, 292, 0, 1),
35    BPF_STMT(BPF_RET | BPF_K, ALLOW),
36    BPF_STMT(BPF_RET | BPF_K, DENY) ;
```

Figure 9: **Classic BPF (cBPF) Program**. Compiled by `sysfilter`, enforcing the following syscall set: 0 (read), 1 (write), 2 (open), 3 (close), 8 (lseek), 9 (mmap), 11 (munmap), 56 (clone), 61 (wait4), 79 (getcwd), 96 (gettimeofday), 102 (getuid), 115 (getgroups), 202 (futext), 292 (dup3), and 317 (seccomp). The filter checks if the value of field `nr` $\in \{0, 1, 2, 3, 8, 9, 11, 56, 61, 79, 96, 102, 115, 202, 292, 317\}$ via means of (deterministic) *skip list*-based search. The `BPF_JEQ` statements assert if the value of `nr` is one of the 16 allowed syscalls, whereas `BPF_JGE` statements implement the “shortcuts” in the search process.